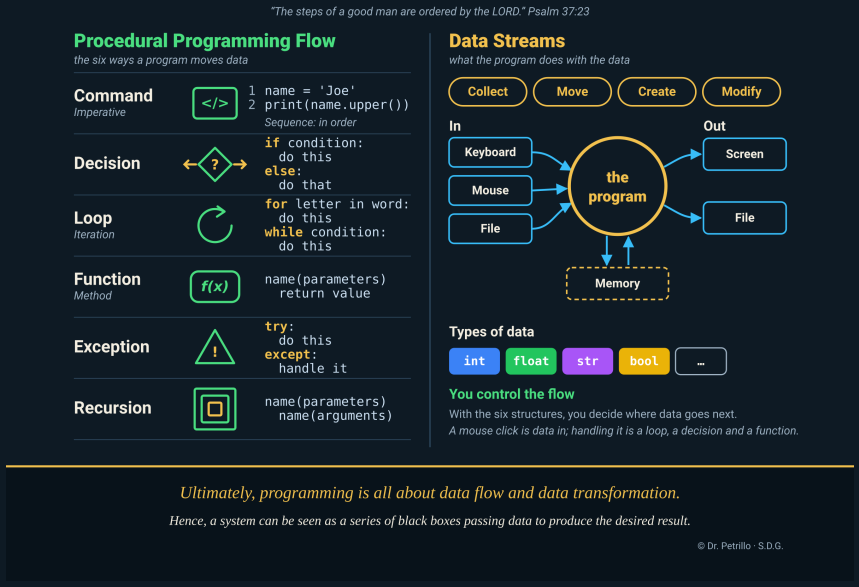


Ultimately, programming is all about data flow and data transformation.

Hence, a system can be seen as a series of black boxes passing data to produce the desired result.

© Dr. Petrillo · S.D.G.

Ordered Steps

Dr. Anthony J. Petrillo, II

Ordered Steps

Dr. Anthony J. Petrillo, II
"Baba Computer"

© Dr. Anthony Petrillo, 2026 · S.D.G.

Published in Africa by Valor. See Baba.Computer for other locations.

All rights reserved. To be used in its entirety.

CONTENTS

Using AI in This Course **1**
- Before you begin . . . 1
- Why so strict? . . . 1
- What you may and may not do . . . 2
- Looking up syntax . . . 2
- Practicing with AI . . . 3
- Your Sphinx site . . . 3
- Your AI log . . . 3

Your Environment **5**
- What you will be using . . . 5
- Wait, AI does this one? . . . 7
- Claude Code is strong. Keep it in its place . . . 8
- No AI log yet . . . 8
- Step 1: Get to a command line, and get Claude Code . . . 8
- Step 2: Let Claude Code set up the rest . . . 9
- Step 3: Learn your way around . . . 9
- What you must be able to do yourself . . . 10
- Words for your glossary . . . 11
- Nothing magical . . . 11

Your Own Site **13**
- What you are building . . . 13
- AI sets this one up too . . . 13
- Let AI walk you through it . . . 13
- What you must be able to do yourself . . . 14
- Your first glossary entry . . . 14

Eggs in a Carton **15**
Is programming necessary? . . . 15
How you will learn . . . 15
Tina's Corner Shop . . . 16
Your egg carton . . . 17
The first moves . . . 17
Tina's first sale . . . 19
The order of the steps . . . 20
Why start with eggs? . . . 21
Your picture so far . . . 21
Exercises . . . 22
Practicing with AI . . . 24
Words for your glossary . . . 24
Summary . . . 24
Answers . . . 25

Say Something **27**
Getting Python ready . . . 27
Tina's sign . . . 27
What the lines say . . . 28
One line after another . . . 29
A few more things `print` can do . . . 29
When it goes wrong . . . 30
The memory game, by hand . . . 34
Your picture so far . . . 35
Exercises . . . 36
Practicing with AI . . . 37
Words for your glossary . . . 38
Summary . . . 38
Answers . . . 39

A Variable Is a Place **41**
From cups to code . . . 41
The assignment statement . . . 42
Names . . . 43
Four kinds of value . . . 43
Working with numbers . . . 45
Working with strings . . . 47
The memory game, on paper . . . 48

Two things a program does with data . . . 50
Your picture so far . . . 50
Exercises . . . 51
Practicing with AI . . . 52
Words for your glossary . . . 52
Summary . . . 52
Answers . . . 53

Data Comes In **55**
Two doors . . . 55
input() . . . 55
Input is always a word . . . 56
Turning a word into a number . . . 58
Showing money properly . . . 59
The receipt . . . 60
Design: input, process, output . . . 61
The memory game: the printed template . . . 62
Your picture so far . . . 65
Exercises . . . 66
Practicing with AI . . . 67
Words for your glossary . . . 67
Summary . . . 67
Answers . . . 68

Choose **73**
How would you decide? . . . 73
True and false . . . 73
if . . . 75
if and else . . . 76
if, elif, else . . . 76
and, or, not . . . 77
Design: pseudocode . . . 78
The shop, version 4 . . . 79
The memory game: a decision . . . 82
Your picture so far . . . 84
Exercises . . . 84
Practicing with AI . . . 86
Words for your glossary . . . 86
Summary . . . 86

Answers 87

Again, While It's True **91**
Sam sweeps the shop 91
while 92
Three kinds of variable a loop needs 93
The loop that never ends 95
The memory game: the Stack begins 95
The memory game moves to the spreadsheet 98
Design: stepwise refinement 99
Your picture so far 105
Exercises 105
Practicing with AI 107
Words for your glossary 107
Summary 107
Answers 108

When Things Go Wrong **113**
A customer types "two" 113
Three kinds of errors 114
Reading a traceback 115
What an exception is 116
try and except 116
Keep asking 118
Playing the memory game with an exception 120
Fixing the shop 122
Design: what could go wrong? 126
Grace at the counter 127
Your picture so far 127
Exercises 127
Practicing with AI 129
Words for your glossary 129
Summary 129
Answers 130

One for Each **133**
A line for every loaf 133
`while` or `for`? 134
Counting with `range` 134
A word is letters in a row 136

Playing the memory game with `for` . . . 137
The memory game moves to the computer . . . 139
Words: four worked examples . . . 140
Tina's receipt . . . 143
Your picture so far . . . 148
Exercises . . . 149
Practicing with AI . . . 150
Words for your glossary . . . 150
Summary . . . 150
Answers . . . 151

Files Remember **155**
The shop forgets . . . 155
Writing: data goes out . . . 156
Reading: data comes in . . . 157
Watching the ▶ move . . . 158
When the file is not there . . . 161
Print statements: the smallest step from the sheet . . . 161
Tina's shop remembers . . . 163
Write it down . . . 168
Your picture so far . . . 169
Exercises . . . 169
Practicing with AI . . . 170
Words for your glossary . . . 170
Summary . . . 171
Answers . . . 171

Name a Job **175**
Your first function . . . 175
Jump away, do a job, come back . . . 177
The door: parameters . . . 178
Sending data back: return . . . 179
Each job has its own cells . . . 180
Design: top-down . . . 183
The shop, rebuilt from named jobs . . . 185
Playing the game with functions . . . 190
Your picture so far . . . 191
Practicing with AI . . . 191
Exercises . . . 192

Words for your glossary . . . 193
Summary . . . 193
Answers . . . 194

A Job That Uses Itself **197**
Closing time . . . 197
Show yourself . . . 198
Work on the way back . . . 200
Words: a job on a smaller word . . . 201
When it never stops . . . 204
Checking your trace with the debugger . . . 204
Your picture so far . . . 205
Practicing with AI . . . 206
Exercises . . . 206
Words for your glossary . . . 207
Summary . . . 208
Answers . . . 208

The Whole Picture **211**
The finished shop . . . 211
A tour: the picture against the program . . . 213
Where are you on the picture? . . . 215
The major project . . . 216
Practicing with AI . . . 218
Exercises . . . 219
Words for your glossary . . . 220
Summary . . . 220
Answers . . . 220
What comes next . . . 221

Syntax Card **223**
The six structures . . . 223
Output and input . . . 226
Comparing and combining (4) . . . 226
Arithmetic (2) . . . 227
The four base types (2) . . . 227
Strings (2, 7, 10) . . . 228
Files (8) . . . 228
Errors you will meet . . . 230

Before you begin

This book assumes you have read *Original Intelligence*. If you have not, read the chapter "Working with AI" before you go on. You need its Freedom Ladder:

Rung	What it means
5. Guiding	You direct the machine and catch its mistakes
4. Verifying	You check what the machine says
3. Comprehending	You can explain it in your own words, with the machine turned off
2. Acquiring	You ask the machine to teach you
1. Doing	The machine does it for you

In this course you may use AI on rung 2, to look things up and to get practice. Rung 3 is your job. Nobody can comprehend for you. Rung 1 is not allowed for anything this course teaches. The next two chapters show the one place it is allowed, setting up your tools, and why.

Why so strict?

Is AI not able to write these programs? Yes, easily. So why learn to write them yourself?

Because you cannot check what you do not understand. Later in your studies you will guide AI and check its work. That is rungs 4 and 5. You cannot stand on them without rung 3 under your feet.

There is an old rule in my classes: *If you cannot solve the problem by yourself, you do not understand. You may understand when someone explains, but that does not count.* AI is a very good explainer. That is exactly why it does not count.

What you may and may not do

You may ask AI	You may not ask AI
What an error message means. Paste the message, never your program.	To write, finish or fix your program
How a piece of syntax is written (see “Looking up syntax”)	To plan or design your program
What a word means. Then write it in your own words in your glossary.	To fill in your memory game
For practice programs to write yourself (see “Practicing with AI”)	To give you the answer to an exercise
For help running your Sphinx site	To write what goes on the pages of your site

And turn off any AI that finishes your code as you type.

Looking up syntax

You will forget how to write things. Every programmer does. Was it `else if` or `elif`? Does the colon go at the end?

Look at your syntax card first. At the back of this book is one page with a template for each of the six structures:

```
if condition:
    do this
else:
    do that
```

The words in plain type, like *condition* and *do this*, are blanks for you to fill in. Most of the time the card has your answer.

If the card doesn’t have it, you may ask AI, but always start your question with this sentence:

> *Answer only with a template that uses blanks, like the one above.*
> *Do not write code for my problem.*

Then ask your question.

Why the sentence? AI is eager to help. Ask how a loop works and it may write the whole program for you, and then you have learned nothing. The sentence keeps AI a dictionary, not a classmate.

Log it as you ask: kind Syntax, your question, and what you learned.

Practicing with AI

Every chapter has exercises, from easy to hard, with answers. Do them. When you want more practice, AI can make more for you. Ask like this:

> *I need help practicing if statements. Starting simple, give me programs that use if statements that I can write. If I get stuck I'll ask for a hint. Then increase their complexity so I get better at designing programs with decisions.*

You write every program. When you are stuck, ask for a hint, not the answer. Then play the memory game on your program to see whether it does what you think.

Your Sphinx site

You will build a Sphinx site in the next chapter. It holds your glossary and your code for this course. AI may help you run the site: setting it up, fixing a build error, adding a page to the table of contents. What goes on the pages is yours: every glossary entry in your own words, every program written by you.

Your AI log

Keeping an AI log is required. It is a page in your site. It starts when your site does, in *Your Own Site*; the setup before that is not logged.

Fill it in while you use AI, not afterwards. Open your log before you open AI. Copy your question into the log as you ask it. Write what you learned before you close AI. Afterwards you will forget, or remember it better than it was.

Each entry has five lines:

- **Date**, as the heading.

- **Kind:** Error, Syntax, Word, Practice or Tool. These are the five things you may ask AI. If your question does not fit one of them, stop. You probably should not be asking it.
- **My question**, copied exactly.
- **What I learned**, in one sentence of your own words.
- **What I did next.**

It looks like this:

```
### 2026-09-14

- **Kind:** Error
- **My question:** What does `IndentationError:␣
↪expected an indented block` mean?
- **What I learned:** Python needs the lines under an␣
↪`if` pushed in, or it doesn't know they belong to it.
- **What I did next:** Indented line 4, and the␣
↪program ran.
```

Once a week, add one line: *Did AI build me up this week, or do my growing for me?*

Your teacher reads the log. It is graded on honesty and completeness, not on how often you used AI. Using AI a lot will not cost you, and using it little will not earn you anything. Nobody but you can see a question you did not log. Keeping the log honest is a matter of integrity, and integrity matters far beyond this course.

What you will be using

In my classes we all work the same way. Here is what you will use:

What	What it is	Why we use it
Debian Linux, a **Mac**, or another **UNIX**	The operating system: the program that runs your computer and all the other programs on it	Debian is free and runs most of the servers in the world. macOS is a UNIX underneath, so nearly every command works the same way. On Windows you run Debian inside Windows, using **WSL** (Windows Subsystem for Linux)
The command line (**CLI**, command line interface)	A window where you type commands and the computer answers in text. The window is called a *terminal*	It is the most direct way to talk to a computer. It works the same on every machine in the room, and on servers that have no mouse at all
Claude Code	Claude, an AI, working inside your terminal. It can see your computer, run commands and change files, if you let it	It sets up your tools for you, on your computer, today
VS Code	An editor: the program you type your code into	Free, and on every operating system
Python	The programming language of this book	Easier to read than most, so you can think about the steps, not the punctuation
IDLE	A small program that comes with Python, for trying lines and running programs	Simple, with nothing to get in the way

You need a **paid Claude account** for Claude Code. This matters. The free account lets you talk to Claude in a browser, but it does not give you Claude

Code. Get the account before you start.

Wait, AI does this one?

In the last chapter I said AI may not do your work. Now I am going to tell you to let AI set up your computer. Is that not a contradiction?

No. It depends on what you are here to learn.

You are here to learn to program: to think in ordered steps and see what each one does to memory. That is the subject. If AI does it, you have learned nothing, so you do it yourself.

Setting up your computer is not the subject. It is a tool, like the pencil you trace with. Nobody makes you build your own pencil before you may write.

There is a second reason. In my old books I warned: *if you only know how to push buttons, when it changes, you will have to take another course.* Setup steps are exactly those buttons. They change with every new version and every different computer. If I printed them here, they would be wrong by the time you read them. AI can work with the computer in front of you, today.

So the rule is not "AI is bad". The rule is: **know what you are learning.** What you are learning, you do yourself. What only gets you there, AI may do for you. That is rung 1 of the Freedom Ladder, and here it is the right rung.

Am I being a hypocrite, then? Telling you to keep AI out of your work, and then handing it a job? No. I use AI in my own work, and you will too. I am not keeping AI from you. I am teaching you when it builds you up and when it does your growing for you. Knowing the difference, in each new situation, is wisdom, and wisdom is the one thing AI cannot give you. So in this course you practice it: some things you do by hand so that you grow, and some things you hand off so that you have time to grow.

You will make this choice many times after this course.

Claude Code is strong. Keep it in its place

Claude Code is not only a chat. It can create files, change them and run commands on your computer. That is why it can set up your tools so well. It is also why you must keep it in its place:

- **Claude Code sets up your tools. It never writes, fixes or reads your programs.** Everything in the AI policy still holds. Claude Code could write every program in this book in a few seconds, and you would learn nothing.
- **Read before you say yes.** Before Claude Code runs a command or changes a file, it asks you. Read what it wants to do. If you do not know what a command does, ask it: *"What will that command do?"* Then decide.
- **One step at a time.** Do not let it run ten things at once. If something breaks, you want to know which step broke it.

No AI log yet

Your AI log is a page in your site, and you build your site in the next chapter. So you do not log the setup in this chapter. Your log starts in the next chapter, and its first entry is the conversation that builds it.

Step 1: Get to a command line, and get Claude Code

You cannot ask Claude Code to install Claude Code. So for this first step, use Claude in your web browser at claude.ai. Start like this:

> *I am a beginner. Help me get a command line and install Claude Code. My computer is: [Windows / Mac / Linux, and its version]. On Windows I want WSL with Debian. Give me one step at a time. Tell me what each step does. Wait for me to tell you what happened before you give the next step.*

Look at how that question is built:

- **"I am a beginner"** tells AI how to talk to you.

- **"My computer is"** gives AI the context it needs to give you steps that work on your computer.
- **"One step at a time … wait for me"** keeps you in charge. If AI gives you ten steps at once and step three fails, you are lost. One step at a time, you always know where you are.

You are done with this step when you can open a terminal, type `claude`, and Claude Code answers you.

Step 2: Let Claude Code set up the rest

Now Claude Code is in your terminal, and it can see your computer. Start it, and ask:

> *I am a beginner learning to program in Python. Help me set up: Python 3, IDLE, and VS Code with its Python extension. [On Windows: I am in WSL, and VS Code should open my WSL files.] Turn off any AI in VS Code that finishes my code as I type. Do one step at a time. Before each step, tell me what it does and wait for me to say yes.*

When it is done, check it yourself. In the terminal, type:

```
python3 --version
```

You should see something like `Python 3.13.5`. The exact number does not matter, as long as it starts with 3.

When something goes wrong, copy the error message to Claude Code. That is allowed here.

Step 3: Learn your way around

Now you need to be able to use these tools yourself. Here Claude Code is not doing the work; it is your guide, and you do every step. Ask:

> *I am a beginner. Give me a short tour of the command line. Teach me one command at a time: where am I, what is here, how to make a folder, how to move into it and back out, and how to see what is in a file. Do not run the commands for me. Tell*

> *me what to type, wait for me to type it in my own terminal, then I will tell you what I saw.*

Notice the last part: *do not run the commands for me.* Claude Code could run them, and you would watch. Watching is not learning. Your hands type, your eyes read the answer.

Then do the same for Python:

> *Now show me two ways to run Python on the command line. First, typing `python3` with nothing after it, to try lines one at a time. Then, running a file with `python3` and the file's name. Use a one-line program that prints a sentence. Tell me what to type; do not type it for me.*

And for IDLE:

> *Now show me IDLE. How do I open it, what is the Shell window, how do I open a new file, save it, and run it? Tell me what to do one step at a time and wait for me.*

You will notice that Python's `>>>` on the command line and IDLE's Shell window do the same thing: you type a line and Python answers right away. That is a good place to try one line when you are not sure what it does.

What you must be able to do yourself

Before you go on, make sure you can do each of these without AI:

1. Open a terminal and find out which folder you are in.
2. List what is in a folder, make a new folder, move into it, and move back out.
3. Start Python on the command line, type `print("Hello")`, see the answer, and leave Python again.
4. Make a file `hello.py` in VS Code with `print("Hello")` in it, save it, and run it with `python3 hello.py`.
5. Open the same file in IDLE and run it from there.
6. Start Claude Code, and leave it again.

If you cannot do one of these, ask Claude Code to explain that step again, then do it yourself.

Words for your glossary

You build your glossary in the next chapter. Write these down now, in your own words, so you can put them in: *operating system*, *terminal*, *command line*, *editor*, *IDLE*.

Nothing magical

There is nothing magical about this environment. We use it so that we are all the same, and so we can work together well: when you are stuck, the student next to you, and I, can see exactly what you see.

When you go to work somewhere, they will have their own environment. Adapt to it. What you learn in this book is not Debian or VS Code or IDLE. It is how to think in ordered steps, and that goes with you anywhere.

What you are building

You will keep your work for this course in a small website of your own. It holds:

- **your glossary**, every new word in your own words;
- **your programs**, each one you write in this course;
- **your AI log**.

The tool that builds the site is *Sphinx*. You write plain text files. Sphinx turns them into web pages, and this is called *building* the site. This book was made with Sphinx too.

Why a site and not a notebook? Because it grows with you. By the end of the course you will have a record of everything you learned, in your own words, that you can search, share and keep.

AI sets this one up too

As in *Your Environment*, Sphinx is a tool that gets you to the subject, not the subject itself. So Claude Code may set it up for you. That is rung 1, on purpose.

Let AI walk you through it

Start Claude Code in your terminal and ask:

> *I am a beginner. Help me set up a Sphinx site on my computer. The site needs a glossary page, a folder for my programs and a page for my AI log. Do one step at a time. Before each step, tell me what it does and wait for me to say yes.*

Claude Code can see your computer, so you do not need to tell it what you have. Read each step before you say yes.

When something goes wrong, copy the error message to AI. That is allowed here.

Write each AI conversation in your AI log, starting with this one. As soon as your log page exists, make this conversation its first entry: kind Tool.

What you must be able to do yourself

AI sets the site up, but it is yours to run. Before you go on, make sure you can do each of these without AI:

1. Find the folder your site lives in, and the file for each page.
2. Build the site and open it in your browser.
3. Add a word to your glossary and see it on the page.
4. Add a program to your programs folder and show it on a page.
5. Add an entry to your AI log.

If you cannot do one of these, ask AI to explain that step again, then do it yourself. Driving the site is your job, even though you didn't build it. You will do these five things every week.

Your first glossary entry

Add your first words now: the words you wrote down in *Your Environment*, and *Sphinx*. Write what it is in your own words, not mine and not AI's.

Is programming necessary?

The quick and easy answer is no. I wrote that in 1994, and it is even more true today. Ask an AI to write a program for a small shop and it will hand you one in seconds. It will probably even run.

So why are you here? Why learn to do by hand what a machine can do for you?

Think about the person who will use that AI. Someone has to tell it what to build. Someone has to read what it built and say, “That part is wrong.” Someone has to know what the program does with the data when the customer types something strange. A program that runs is not the same as a program that is right.

You can only check what you understand. If you do not know how a program works inside, you are trusting a stranger every time you use AI to write code. If you do know, you can use AI the way a carpenter uses a power saw: fast, and with your eye on the line.

That is what this book is for. Not to make you faster than AI at typing code. To make you the person who understands what the code does, so you can build with it, check it and direct it.

So the honest answer is this. If you will only ever *use* programs, you do not need this book. If you will build things, design systems for other people, or work with AI on anything that matters, you need to know what happens inside the machine. Read on.

How you will learn

I have taught programming since the 1980s, in Nigeria, in Liberia and in the United States. I have watched many students try to learn it by reading. It does not work. Programming is learned the way a child learns to speak: by using it, getting it wrong, and using it again.

Many students come to me struggling in mathematics and ask for help. When

I give them a plan, it always includes practice, an hour a day. They are disappointed. They hoped for some magic that would make them understand. There is no magic. Every discipline is learned by practice, from mathematics to music. It is even true of faith. You can study about God all you want, but if you do not practice what you learn, you have not learned it. Programming is the same. You will practice, and the practice is the learning.

Throughout this book I will ask you questions. They look like this:

⧖ **0.1)** What do you think a computer's *memory* is?

Stop. Answer it yourself, in your notebook or your head, before you read on.

Answer. Memory is where the computer keeps the data it is working on right now: a number, a name, a price. It is made of many small places, and each place can hold one thing. That is all you need for now, and you are about to build one.

Most questions are answered right after they are asked. Do not skip ahead to the answer. The few seconds you spend thinking are where the learning happens.

> **For your glossary:** *program*, *memory*. Write each one in your own words in the glossary on your site.

Tina's Corner Shop

Let me introduce someone you will spend the whole book with.

Tina keeps a small corner shop. She sells three things:

Item	Price	In stock
Bread	2.50 a loaf	12 loaves
Milk	1.75 a carton	20 cartons
Eggs	3.00 a carton of 12	10 cartons

The prices are plain numbers. Use whatever money you use at home.

Tina wants a program to help her run the shop: show her prices, take an order, work out the total, keep track of what is left, and remember the day's sales. You are going to write that program, one piece at a time, through this whole

book. By the end it will do all of that, and you will know what every line of it does.

But we are not going to start at the computer. We are going to start at the table.

Your egg carton

Before the next class, get these:

- **An egg carton.** An empty one, cleaned out. If you cannot get one, use a row of small cups, or cut up a box into sections. Anything with separate places will do.
- **Small slips of paper** that fit inside one cup.
- **Sticky notes or tape** for labels.
- **A pencil.**

The egg carton is going to be the computer's memory. Each cup is one place in memory. A carton of twelve has two rows of six cups: twelve places.

Here are the three rules of the carton. They are the rules of real memory too.

1. **A cup holds one value at a time.** One slip of paper, no more.
2. **Looking in a cup does not empty it.** You can read the slip as often as you like. It stays.
3. **Putting a new value in throws the old one away.** Take the old slip out and tear it up. Then put the new one in.

When you put a label on a cup, you have made a *variable*: a named place in memory. The label is the variable's *name*. The slip inside is its *value*.

> **For your glossary:** *variable*, *value*.

The first moves

Take out your carton. We are going to do what a computer does, one step at a time. Read each step, then do it with your hands. Do not just read. Do it.

Step 1: `x = 1`

Put a label `x` on a cup. Write `1` on a slip and put it in the cup.

That line is called an *assignment*. Read the = sign as **gets**, not "equals". `x = 1` says *x gets 1*.

Step 2: `y = 2`

Label another cup `y`. Put a slip with `2` in it.

Step 3: `z = x + y`

This one has work in it. Do it in this order:

1. Look in `x`. It holds 1.
2. Look in `y`. It holds 2.
3. Add them: 3.
4. Label a new cup `z` and put a slip with `3` in it.

⧖ **0.2)** After step 3, what is in cup `x`?

Answer. Still 1. Rule 2: looking in a cup does not empty it. You read `x` and `y`, you did not take anything out of them.

Notice the order. The computer always works out the **right side** of the = first, and only then puts the answer into the cup named on the **left side**. Remember that. It is about to matter.

Step 4: `x = x + 7`

Stop and read that line as algebra. *x equals x plus 7*. That is nonsense. No number equals itself plus 7.

⧖ **0.3)** So what does the computer do with it?

Answer. Do it with the cups, right side first:

1. Look in `x`. It holds 1.
2. Add 7: 8.
3. Now put 8 into the cup on the left, which is `x`. Rule 3: take the old slip out and tear it up. Put the new slip, `8`, in.

The old value, 1, is gone. Your own hand threw it away.

This is the most important thing in this chapter, so let me say it plainly: **a variable is a place, not a value.** `x` is not "1". `x` is a cup. Right now it holds 8. A moment ago it held 1. The cup stayed the same; what was in it changed.

Students who skip this step struggle for months. Students who do it with their hands do not.

Tina's first sale

Empty your carton. Let's do the same thing with Tina's shop. A customer buys three loaves of bread.

```
price = 2.50
count = 3
total = price * count
```

In Python, the `*` sign means multiply.

Do it with the carton:

1. `price = 2.50`: label a cup `price`, put in a slip with `2.50`.
2. `count = 3`: label a cup `count`, put in `3`.
3. `total = price * count`: look in `price` (2.50), look in `count` (3), multiply: 7.50. Label a new cup `total` and put in `7.50`.

Now Tina's stock. She had 12 loaves. She just sold 3.

```
stock = 12
stock = stock - 3
```

4. `stock = 12`: label a cup `stock`, put in `12`.
5. `stock = stock - 3`: right side first. Look in `stock` (12), subtract 3: 9. Take out the 12, tear it up, put in `9`.

⧖ **0.4)** After step 5, how many cups are labelled `stock`?

Answer. One. The second line did not make a new cup. It used the same cup and changed what was in it. That is exactly what Tina wants: one count of bread, kept up to date.

⧖ **0.5)** Here is a trap. Suppose the customer changes her mind and wants 4 loaves, not 3. You do this:

```
count = 4
```

What is in `total` now?

Answer. Still 7.50. Look in the cup. Nothing told the computer to work out the total again. `total = price * count` was done once, at step 3, with the values that were in the cups at that moment. Changing `count` later does not reach back and change `total`. If Tina wants the new total, the program must say so again:

```
total = price * count
```

Now `total` holds 10.00.

This surprises almost everyone the first time. A spreadsheet would have updated the total for you. A program does not. A program does exactly what its lines say, in order, and nothing else.

The order of the steps

⧖ **0.6)** What happens if the lines come in this order?

```
total = price * count
price = 2.50
count = 3
```

Answer. Try it with the carton. The first line says: look in `price`. There is no cup labelled `price` yet. You cannot look in a cup that does not exist. The computer stops with an error at the very first line.

The order of the lines is the order the work is done. Step one, then step two, then step three. That is the first and simplest way a program moves: in *sequence*, one step after another. Every program you ever write is built on it.

For your glossary: *assignment*, *sequence*.

Why start with eggs?

You may feel this is too simple. Cups and slips of paper, at university?

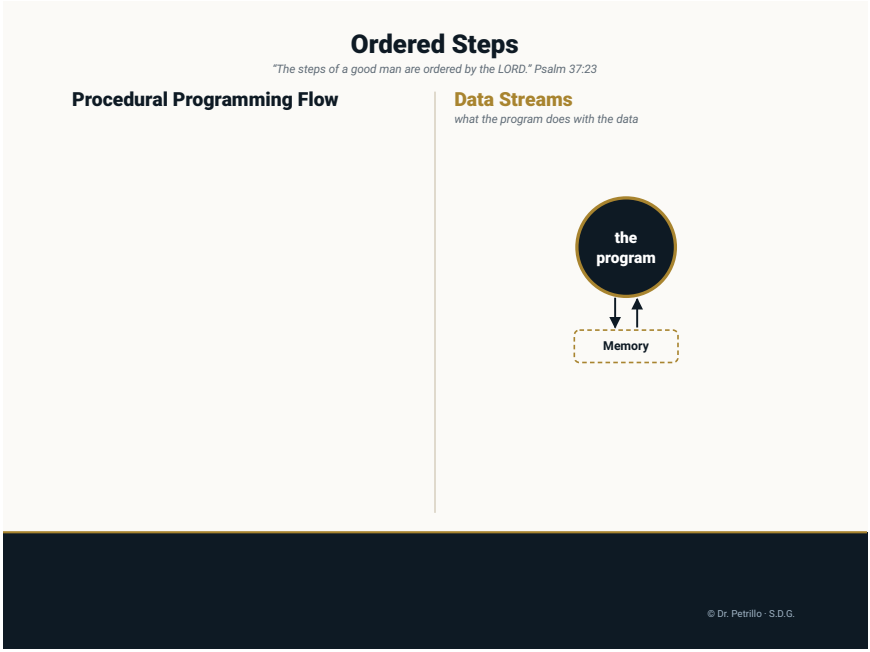
In 1993 I wrote a book for students who did not have computers. Most students in the world did not. We taught the computer with doors, charts and pencils, and those students understood the machine better than many who had one. Years later I brought egg cartons into my programming classes. Students laughed. Then they stopped laughing, because the carton showed them what the screen hides: that every value lives in a place, and that a new value throws the old one away.

Moving your hand to put a slip in a cup does something for your memory that reading does not. And it is your memory, not the computer's, that we are really training.

You will come back to the carton whenever something new arrives in memory. Keep it.

Your picture so far

Over this book you are going to build a picture of how a program works, one piece at a time. Here is the first piece: **memory**, the place where data sits.



Exercises

Do these with your egg carton. For each one, write down what is in every cup at the end.

1. Trace these lines:

```
a = 5
b = 10
```

2. Trace these lines:

```
a = 5
b = a
a = 9
```

   What is in `b` at the end?

3. Milk costs 1.75. A customer buys 2 cartons. Trace:

```
milk_price = 1.75
count = 2
total = milk_price * count
```

4. Trace:

```
eggs = 10
eggs = eggs - 4
eggs = eggs + 6
```

5. Trace:

```
n = 3
n = n * 2
n = n * 2
n = n * 2
```

6. What goes wrong here, and why?

```
stock = stock - 3
stock = 12
```

7. Trace these lines and say what `total` holds at the end. Then say what it *should* hold if Tina wanted the total for the new count.

```
price = 3.00
count = 2
total = price * count
count = 5
```

8. Tina has two cups, `bread_stock` holding 12 and `milk_stock` holding 20. She labelled them the wrong way round, and wants to swap their values. Using the carton, find a way to swap them so that `bread_stock` holds 20 and `milk_stock` holds 12. Write down your steps as assignment lines. (Hint: you may need one more cup.)

9. Write your own list of five assignment lines for a sale of eggs: the price, the count, the total, the stock, and the stock after the sale. Trace it with your carton and check it.

Practicing with AI

When you want more practice, ask like this:

> *I need help practicing tracing assignment statements by hand. Starting simple, give me short lists of Python assignment lines, like* `x = 3` *and* `x = x + 2`*, that I can trace with an egg carton. Do not give me the answers. If I get stuck I'll ask for a hint. Then increase their complexity so I get better at following what happens in memory.*

Log it in your AI log as you ask: kind Practice, your question, and what you learned.

Words for your glossary

Write each of these in your own words in your site's glossary: **program**, **memory**, **variable**, **value**, **assignment**, **sequence**.

Summary

- **Is programming necessary?** To use programs, no. To build them, check them and direct AI, yes: you can only check what you understand.
- **Memory** is made of places. Each place holds one value.
- A **variable** is a named place in memory. It is a place, not a value.
- **Assignment** (=, read as *gets*): work out the right side first, then put the answer in the variable on the left.
- The carton's three rules: a cup holds one value; looking does not empty it; a new value throws the old one away.
- A program does exactly what its lines say, **in sequence**, and nothing else. A total worked out earlier does not update itself.

Answers

1. `a` holds 5, `b` holds 10.
2. `a` holds 9, `b` holds 5. The line `b = a` copied the 5 into `b`'s cup. Changing `a` later did not change `b`.
3. `milk_price` 1.75, `count` 2, `total` 3.50.
4. `eggs` holds 12: 10, then 6, then 12. Still one cup.
5. `n` holds 24: 3, 6, 12, 24.
6. The first line looks in `stock` before any cup has that label. The computer stops with an error. The lines are in the wrong order: `stock = 12` must come first.
7. `total` holds 6.00, from the count of 2. For a count of 5 it should hold 15.00, but only if the line `total = price * count` is done again after `count = 5`.
8. One way:

   ```
   spare = bread_stock
   bread_stock = milk_stock
   milk_stock = spare
   ```

   `spare` keeps a copy of 12 safe while `bread_stock` gets 20. Without the third cup, the first assignment would throw away the 12 for good. Try it without `spare` and watch it go wrong.
9. Answers will differ. One example:

   ```
   egg_price = 3.00
   count = 2
   total = egg_price * count
   egg_stock = 10
   egg_stock = egg_stock - count
   ```

   At the end: `egg_price` 3.00, `count` 2, `total` 6.00, `egg_stock` 8.

In the last chapter you were the computer, with a carton and slips of paper. Now we bring in the real one. Its first job for Tina is a small one: put up her shop sign.

Getting Python ready

You set up Python and VS Code in *Your Environment*. Check that Python is still there. In a terminal, type:

```
python3 --version
```

You should see something like `Python 3.13.5`. The exact number does not matter, as long as it starts with 3.

Two more things before you start:

- **Make sure the AI in your editor is off,** the kind that finishes your code as you type. You are learning to write it.
- **Use the programs folder in your site,** the one you set up in *Your Own Site*. Every program you write in this book goes there.

You can write and run your programs in VS Code and the terminal, or in IDLE. Both run the same Python. In this book I show the terminal.

Tina's sign

Make a new file called `shop.py` in your programs folder. Type this into it exactly:

```
print("Tina's Corner Shop")
print("Open 8 to 6")
print("Bread 2.50  Milk 1.75  Eggs 3.00")
```

Save it. Then run it. In a terminal, in the folder where `shop.py` is, type:

```
python3 shop.py
```

You should see:

```
Tina's Corner Shop
Open 8 to 6
Bread 2.50  Milk 1.75  Eggs 3.00
```

That is your first program. It is small, but it is a real program, and everything you write from now on grows out of it.

Keep a copy of it on your site as `shop_ch01.py`. You will keep each chapter's version of the shop, so you can look back and see how it grew.

What the lines say

Look at one line:

```
print("Open 8 to 6")
```

- `print` is a command Python already knows. It means: show this on the screen.
- The round brackets, `(` and `)`, hold what you want shown.
- The double quotes, `"` and `"`, mark the start and end of the text. Text in quotes is called a *string*: a string of characters, one after another.

The quotes are not shown on the screen. They only tell Python where the text starts and stops.

> **For your glossary:** *string*.

⧖ **1.1)** Python lets you use either double quotes `"…"` or single quotes `'…'` around a string. Why does Tina's first line use double quotes?

Answer. Because of the apostrophe in *Tina's*. An apostrophe is the same character as a single quote. If you wrote `'Tina's Corner Shop'`, Python would think the string ended right after `Tina`, and the rest would make no sense to it. With double quotes around the outside, the apostrophe inside is just a character.

One line after another

⧖ **1.2)** What does this program show?

```
print("Open 8 to 6")
print("Bread 2.50  Milk 1.75  Eggs 3.00")
print("Tina's Corner Shop")
```

Answer.

```
Open 8 to 6
Bread 2.50  Milk 1.75  Eggs 3.00
Tina's Corner Shop
```

The shop name is now at the bottom. Python does the lines in the order they are written, top to bottom, one at a time. It does not know that a shop name belongs at the top of a sign. It does what the lines say, in the order they say it.

You met this in the last chapter as *sequence*. It is the first of the six ways a program moves, and it has a name: a **command**. A command says *do this one thing, then go to the next line*. Every line you have written so far is a command. Most lines you will ever write are.

> **For your glossary:** *command*.

A few more things `print` can do

A blank line:

```
print()
```

`print` with nothing in the brackets shows an empty line. Use it to space out what you show.

More than one thing at once:

```
print("Bread", "2.50", "Milk", "1.75")
```

shows

```
Bread 2.50 Milk 1.75
```

Separate things with commas, and `print` puts one space between them.

A *comment*:

```
# Tina's shop sign
print("Tina's Corner Shop")
```

A line that starts with `#` is a comment. Python skips it completely. It is a note for people, not for the computer. Do not comment the obvious; `# print the shop name` above a line that prints the shop name helps nobody. Comment when something is hard, or when you will need to remember *why* you did something.

For your glossary: *comment*.

Put it all together:

```
# Tina's shop sign
print("Tina's Corner Shop")
print()
print("Open 8 to 6")
print("Bread", "2.50", "Milk", "1.75")
```

```
Tina's Corner Shop

Open 8 to 6
Bread 2.50 Milk 1.75
```

When it goes wrong

It will go wrong. That is normal, and it is how you learn.

When I go to the market in Jos, I greet people and bargain in broken Hausa. My Hausa is nowhere near correct, but people understand me and I come home with what I need. People can find the meaning in broken speech.

A computer cannot. It is a very simple machine. It has none of the flexibility of a person, and it will not guess what you meant. So in programming there

are two kinds of being right:

- **Syntax** is the rules for writing a line: where the brackets go, where the quotes go, how words are spelled.
- **Semantics** is the meaning: whether the line does what you wanted.

With people, you can get the meaning across with poor syntax. With a computer, you must get the syntax exactly right before you can even find out whether the meaning is right.

For your glossary: *syntax*, *semantics*.

A missing quote

Here is the sign with one quote missing on line 2:

```
print("Tina's Corner Shop")
print("Open 8 to 6)
print("Bread 2.50  Milk 1.75  Eggs 3.00")
```

Run it, and Python shows:

```
  File ".../shop.py", line 2
    print("Open 8 to 6)
          ^
SyntaxError: unterminated string literal (detected at␣
↪line 2)
```

(Where you see `...`, your screen shows the folder your file is in.)

Read it slowly. It tells you the file, the line (2), shows you the line, and puts a `^` under the place where it got confused. Then it names the problem: a *syntax error*. "Unterminated string literal" means a string that never ends: you opened a quote and never closed it.

⧖ **1.3)** Look at what was printed. Did the first line, `Tina's Corner Shop`, show up?

Answer. No. Nothing was shown at all. Before Python does a single line, it reads the whole file to check the syntax. If any line breaks the rules, it refuses to start.

A misspelled command

Now a different mistake, a capital letter on line 2:

```
print("Tina's Corner Shop")
Print("Open 8 to 6")
print("Bread 2.50  Milk 1.75  Eggs 3.00")
```

```
Tina's Corner Shop
Traceback (most recent call last):
  File ".../shop.py", line 2, in <module>
    Print("Open 8 to 6")
    ^^^^^
NameError: name 'Print' is not defined. Did you mean:
↪'print'?
```

⧖ **1.4)** This time the first line *was* printed. Why the difference?

Answer. `Print("Open 8 to 6")` follows the syntax rules. It has a name, brackets and a closed string, so Python's check passes and the program starts. Python does line 1. Then it reaches line 2 and looks for a command called `Print`. There is none: Python is *case sensitive*, and `Print` with a capital P is a different word from `print`. So it stops there, in the middle of the run. Line 3 is never reached.

That is sequence again. The program runs one line after another until it hits trouble, and it stops on the line where the trouble is.

> **For your glossary:** *case sensitive*.

Watch closely

Here are the mistakes I see most often from beginners:

Mistake	What is wrong
`Print("Op`	Capital letter. Python is case sensitive.
`print("Op`	A quote opened but never closed.
`print("Op`	A bracket opened but never closed. Python says `'(' was never closed`.
`print('Ti Corner Shop')`	The apostrophe ends the string early. Use double quotes outside.
`print("O`	Spaces at the start of a line. Python says `IndentationError: unexpected indent`. Spaces at the start mean something in Python, as you will see in chapter 4. For now, start every line at the left edge.
`print("Op`	Curly "smart" quotes, copied from a word processor. Python only knows straight quotes.

Python usually tells you the line where it noticed the problem. Sometimes the real mistake is on the line *before*, such as a bracket left open. Look there too.

A bug

⧖ **1.5)** Tina's shop is open from 8 to 6. Her program says:

```
print("Open 8 to 5")
```

Is that a syntax error?

Answer. No. The line is perfectly good Python. It runs without a complaint. It is just *wrong*: it tells customers the shop closes an hour early. That is a *semantic error*, usually called a *bug*. Python can catch your syntax for you. It can never catch your meaning. Only you can, by checking what the program does against what it should do.

The word *bug* has a story. In 1947, Grace Hopper and the team working on the Harvard Mark II, a computer the size of a room, found it failing. The cause was a moth caught in one of its switches. They taped the moth into their logbook with the note "First actual case of bug being found." Engineers already called faults bugs, but that moth made the word famous, and errors in programs have been called bugs ever since.

For your glossary: *syntax error*, *bug*.

Error messages and AI

Error messages are one of the places you may use AI. If you do not understand a message, copy **the message**, not your program, and ask what it means. Then fix the program yourself. Log it as you ask: kind Error.

The memory game, by hand

Now the most important part of this chapter.

In the last chapter you were the computer with an egg carton. Now you will be the computer on paper. This is the *memory game*. The name is a play on words: it is the computer's memory, and it is yours.

Here is how it works. You take one line of the program. You copy it onto a sheet of paper. Then you do exactly what the computer does with that line, no more and no less, and you write down everything that changed. Then you take a fresh sheet and do the next line.

One sheet per line. The sign program has three lines, so it takes three sheets.

I am not going to give you a form. Take a blank sheet of paper and decide for yourself what you need to write down. Ask yourself:

- How will I know which line this sheet is for?
- Where do I copy the line?
- Where do I show what is on the screen so far?

Draw it. You will get the layout wrong. That is on purpose. Getting it wrong is what makes the form you will be handed later make sense: every box on it will answer a question you have already asked.

Do it now, for the three lines of Tina's sign, before you read on.

⧖

⧖ **1.6)** When you got to the third sheet, what did you write for the screen: only the third line of the sign, or all three?

Answer. All three. The screen does not clear itself between lines. Each `print` adds a line below what is already there. So by the third sheet the

screen shows the whole sign. If your sheet only showed one line, you have just found out something about the screen.

Most students end up with something like this for the third sheet:

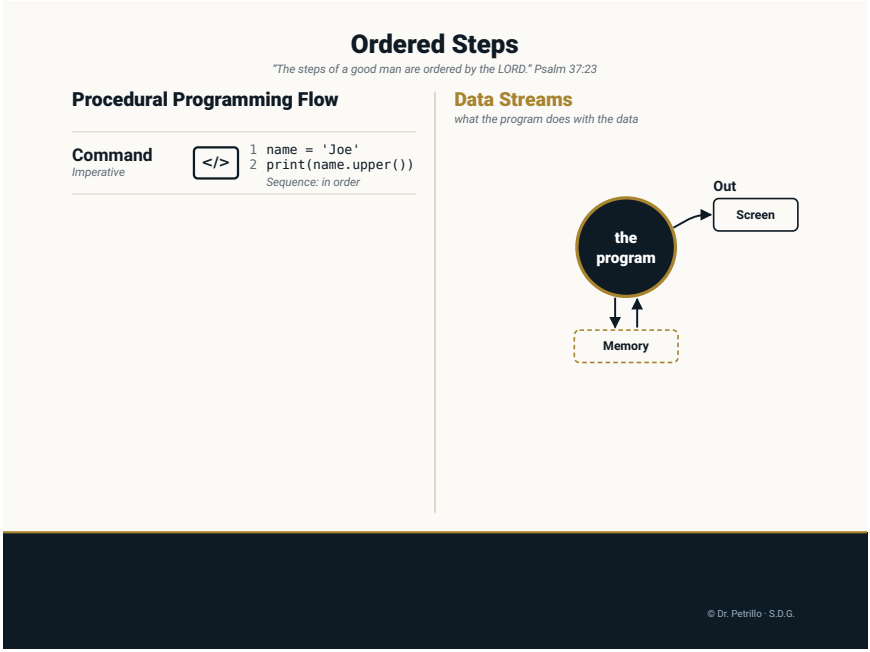
```
Line: 3
Code: print("Bread 2.50  Milk 1.75  Eggs 3.00")
Kind: Command
Screen:
    Tina's Corner Shop
    Open 8 to 6
    Bread 2.50  Milk 1.75  Eggs 3.00
```

Yours may look different. That is fine. Keep your sheets. As the book goes on, you will need more places on the sheet: one for memory in the next chapter, one for what is typed on the keyboard after that. You will add them as you need them.

For your glossary: *memory game*.

Your picture so far

Two new pieces: the **command**, the first of the ways a program moves, and the **screen**, the first way data goes out.



Exercises

1. Write a program that shows your name on one line and your town on the next.

2. What does this show? Work it out before you run it.

```
print("Eggs 3.00")
print()
print("Milk 1.75")
print("Bread 2.50")
```

3. What does this show?

```
print("Bread", "2.50")
print("Milk", "1.75", "Eggs", "3.00")
```

4. Find and fix the mistake in each line:

```
print("Fresh bread today)
PRINT("Milk 1.75")
print "Eggs 3.00"
print('Tina's prices')
```

5. For each, say whether it is a syntax error or a bug:
    1. `print("Bred 2.50")` on Tina's sign.
    2. `print("Bread 2.50"`
    3. `print("Eggs 30.00")` when eggs cost 3.00.
    4. `pritn("Eggs 3.00")`
6. This program has a mistake on line 3. What is shown before it stops?

```
print("Welcome")
print("Bread 2.50")
prnt("Milk 1.75")
print("Eggs 3.00")
```

7. Play the memory game by hand for the program in exercise 2. How many sheets do you need? What is on the screen on the last sheet?
8. Tina wants a box of stars around her shop name. Write a program that shows exactly this:

```
************************
*  Tina's Corner Shop  *
************************
```

9. Write a sign for a shop of your own: its name, its hours, and three things it sells with prices. Add one comment that says something a reader would not guess from the code. Play the memory game on it.

Practicing with AI

I need help practicing print statements and finding syntax errors in Python. Starting simple, give me short programs that use only print, some with mistakes in them, that I can work out by hand. Do not tell me where the mistakes are. If I get stuck I'll ask for a

hint. Then increase their complexity so I get better at reading and fixing code.

Log it as you ask: kind Practice.

Words for your glossary

Write each of these in your own words in your site's glossary: **string**, **command**, **comment**, **syntax**, **semantics**, **case sensitive**, **syntax error**, **bug**, **memory game**.

Summary

- **Python** runs the lines in a file. Run `shop.py` with `python3 shop.py`.
- `print("text")` shows text on the screen. `print()` shows a blank line. Commas put a space between things.
- A **string** is text in quotes. Use double quotes when the text has an apostrophe.
- A **comment** starts with `#`. Python skips it.
- A **command** does one thing, then the program goes to the next line. Lines run in **sequence**, top to bottom.
- **Syntax error**: Python refuses to start; nothing runs. Other errors stop the program on the line where they happen, after the lines before it have run.
- **Bug**: the program runs, but it is wrong. Only you can catch it.
- Python is **case sensitive**.
- **The memory game**: one sheet per line run. Copy the line, do what it says, write down what changed. Draw your own sheet first.

Answers

1. For example:

```
print("Ruth Okafor")
print("Jos")
```

2.
```
Eggs 3.00

Milk 1.75
Bread 2.50
```

3.
```
Bread 2.50
Milk 1.75 Eggs 3.00
```

4.
```
print("Fresh bread today")
print("Milk 1.75")
print("Eggs 3.00")
print("Tina's prices")
```

   The first was missing a closing quote; the second used capitals; the third was missing the brackets; the fourth needed double quotes because of the apostrophe.

5. 1. Bug: it runs, but bread is misspelled. 2. Syntax error: the bracket is never closed. 3. Bug: it runs, but the price is wrong. 4. Not a syntax error, but the program stops when it reaches that line, because there is no command called `pritn`. It is a mistake Python catches while running.

6.
```
Welcome
Bread 2.50
```

   Then it stops with `NameError: name 'prnt' is not defined`. Line 4 never runs.

7. Four sheets, one per line. On the last sheet the screen shows:

```
Eggs 3.00

Milk 1.75
Bread 2.50
```

The blank line counts: line 2's sheet shows the screen with an empty line under `Eggs 3.00`.

8. ```
   print("************************")
   print("*  Tina's Corner Shop  *")
   print("************************")
   ```

   Count the characters so the stars line up: the middle line is 24 characters, so each row of stars is 24 stars.

9. Answers will differ. Check that it runs, that every line on the screen is what you meant, and that your sheets match what the program shows.
   ```

Tina's sign shows her prices, but the program does not *know* them. They are just text on a sign. If bread goes up to 2.75, you have to find every place you typed `2.50` and change it by hand. And the program cannot work anything out with a price that is only text.

In this chapter the prices, the stock and the shop's name move into memory, into the cups you met in chapter 0. Then the program can work with them.

From cups to code

Here is the next version of `shop.py`:

```
shop = "Tina's Corner Shop"
bread_price = 2.50
bread_stock = 12
is_open = True
print(shop)
total = bread_price * 3
bread_stock = bread_stock - 3
print("Bread left:", bread_stock)
```

Before you run it, take out your egg carton and do it by hand, one line at a time, the way you did in chapter 0.

⧖

⧖ **2.1)** How many cups did you use?

Answer. Five: `shop`, `bread_price`, `bread_stock`, `is_open` and `total`. Line 7 did not need a new cup. It changed what was in `bread_stock`.

Now run it:

```
Tina's Corner Shop
Bread left: 9
```

Line 5, `print(shop)`, has no quotes around `shop`. That matters. With

quotes, `print("shop")` would show the word *shop*. Without quotes, Python looks in the cup named `shop` and shows what is in it.

Keep this version on your site as `shop_ch02.py`.

The assignment statement

Every line in that program except the two `print` lines is an *assignment statement*:

```
name = value
```

You already know how to read it from chapter 0: the name on the left **gets** the value on the right. And you know the order: **work out the right side first, then put the answer in the cup on the left.**

That order is why this line works:

```
bread_stock = bread_stock - 3
```

Right side first: look in `bread_stock` (12), subtract 3, get 9. Then put 9 in `bread_stock`, throwing the 12 away.

Do it once more with the carton if it still feels strange. It is the line that turns a variable from a word into a place you can see.

The first time a name is given a value, Python makes the cup. Every time after that, it uses the same cup.

⧖ **2.2)** What does this show?

```
bread_stock = 12
bread_stock = 20
print(bread_stock)
```

Answer. `20`. One cup. The 12 was put in, then thrown away when the 20 went in.

> **For your glossary:** *assignment statement*.

Names

The name of a variable is called an *identifier*. Python has rules for them:

- Letters, digits and the underscore _ only. No spaces, no dashes.
- It may not start with a digit.
- Capitals matter: `Price` and `price` are two different cups.
- It may not be one of Python's own words, such as `if`, `while`, `for` or `True`.

And there are habits good programmers keep:

- **Say what it holds.** `bread_price` tells you something. `bp` tells you nothing, and in a month you will not remember what it meant.
- **Use small letters with underscores between words**: `bread_price`, `is_open`. That is the Python habit.

⧖ **2.3)** Which of these names are allowed?

```
bread price    bread_price    2nd_price    price2    ␣
↪Price    if
```

Answer. `bread_price`, `price2` and `Price` are allowed. `bread price` has a space. `2nd_price` starts with a digit. `if` is one of Python's own words. If you try `bread price = 2.50`, Python stops before it starts:

```
    bread price = 2.50
          ^^^^^
SyntaxError: invalid syntax
```

For your glossary: *identifier*.

Four kinds of value

Look at the four values Tina's program puts in memory:

Variable	Value	Kind
`shop`	`"Tina's Corner Shop"`	text
`bread_price`	`2.50`	a number with a decimal point
`bread_stock`	`12`	a whole number
`is_open`	`True`	yes or no

Each kind of value is called a *type*. These are the four *base types* of Python, the ones everything else is built from:

Type	Python's name	What it holds	Examples
Integer	`int`	whole numbers	`12`, `0`, `-3`
Floating point	`float`	numbers with a decimal point	`2.50`, `1.75`, `0.5`
String	`str`	text	`"Bread"`, `"Tina's Corner Shop"`
Boolean	`bool`	true or false	`True`, `False`

Why does the type matter? Because it decides what you can do with the value. You can multiply a price. You cannot multiply a shop name by a shop name. And as you will see, the same symbol does different things to different types.

You can ask Python what type a value is:

```
print(type(12))
print(type(2.50))
print(type("Tina's Corner Shop"))
print(type(True))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

Ignore the word `class` for now; it is Python's word for a type. What you want is the name in quotes.

A few things to watch:

- `True` and `False` start with a capital letter. `true` is not a value; Python will say `NameError: name 'true' is not defined. Did you mean: 'True'?`
- A *boolean* is named after George Boole, a teacher in the 1800s who worked out the mathematics of true and false. Every decision a computer makes rests on it. You will use booleans in chapter 4.
- `"12"` in quotes is a string, not a number. It is the characters *1* and *2*, the way they look on a sign.

  For your glossary: *type*, *integer*, *float*, *string*, *boolean*.

⧖ **2.4)** What does this show?

```
print(2.50)
```

Answer. `2.5`. A float does not keep a zero at the end; 2.50 and 2.5 are the same number. That is fine for working things out, but not for a price on a receipt. In chapter 3 you will learn to show a number with exactly two decimal places.

Working with numbers

Python's arithmetic signs:

Sign	Meaning	Example	Result
`+`	add	`12 + 3`	`15`
`-`	subtract	`12 - 3`	`9`
`*`	multiply	`2.50 * 3`	`7.5`
`/`	divide	`7 / 2`	`3.5`
`//`	divide, whole part only	`7 // 2`	`3`
`%`	remainder	`7 % 2`	`1`
`**`	power	`2 ** 3`	`8`

⧖ **2.5)** What is `6 / 3`?

Answer. `2.0`, not `2`. Plain division `/` always gives a float, even when it comes out even.

The two odd ones, `//` and `%`, are more useful than they look. Tina gets a delivery of 125 eggs, and she sells them in cartons of 12.

⧖ **2.6)** How many full cartons can she fill, and how many eggs are left over?

Answer.

```
eggs = 125
cartons = eggs // 12
left_over = eggs % 12
print(cartons, left_over)
```

```
10 5
```

Ten full cartons, and five eggs left over. `//` gives how many whole times 12 goes into 125. `%` gives what remains. In Nigeria I gave students the same problem with grain: 2,560 kilograms into 50 kilogram bags. Every shop and farm has this problem.

Which comes first

⧖ **2.7)** A customer buys 3 loaves and 2 cartons of milk. What does this show?

```
print(2.50 * 3 + 1.75 * 2)
```

Answer. `11.0`. Python does multiplication and division before addition and subtraction, just as in mathematics: 7.5 plus 3.5. This order is called *precedence*.

When you want a different order, use brackets. Brackets are always done first:

```
print(2 + 3 * 4)
print((2 + 3) * 4)
```

```
14
20
```

If you are not sure what Python will do first, use brackets. They cost nothing and they make your meaning plain to the next person who reads your code.

For your glossary: *precedence*.

A float is close, not exact

Try this:

```
print(0.1 + 0.2)
```

```
0.30000000000000004
```

That is not a mistake in Python. A float is stored in a fixed number of binary digits, and most decimal fractions cannot be written exactly in binary, just as one third cannot be written exactly in decimal (0.3333…). So a float is very close, but not always exact. For a shop it is close enough, as long as you show prices rounded to two places, which you will learn in chapter 3.

It is a good reminder of something worth remembering about every machine people build: it is limited. It is very good, and very fast, and still limited. Trust it, and check it.

Working with strings

The same + sign does something different with strings. It joins them:

```
print("Tina" + "'s")
```

```
Tina's
```

⧖ **2.8)** What does each line show?

```
print("12" + "12")
print(12 + 12)
```

Answer.

```
1212
24
```

The first joins two strings, the characters *1 2* and *1 2*. The second adds two numbers. Same sign, different types, different work. This is why the type matters.

⧖ **2.9)** What happens here?

```
print("Bread left: " + 9)
```

Answer. Python stops:

```
TypeError: can only concatenate str (not "int") to str
```

Concatenate means join. Python can join a string to a string, but not a string to a number. It will not guess which you meant. Use a comma instead, which shows each thing with a space between, as you did in the program:

```
print("Bread left:", 9)
```

The memory game, on paper

Now you need a new place on your memory game sheet: **memory**.

Take out your sheets from chapter 1. Add a region for memory. It should look like your egg carton: a row of boxes, one per variable. Give each box an *address*, the way a spreadsheet names its cells: `A1`, `B1`, `C1` and so on along the row. The address is where the variable lives. The name is what the program calls it.

You also need a small table to keep track of which name lives at which address, and what type it holds. Call it the *data dictionary*. It has three columns for now:

Name	Type	Location

You will add more columns later in the book, when you need them.

Play the memory game on the shop program above: one sheet per line, eight sheets. On each sheet, write the line, write *Command* for the kind of line, update memory, update the data dictionary, and update the screen.

⧖

Here is what sheet 7 should hold, for `bread_stock = bread_stock - 3`:

```
Line: 7
Code: bread_stock = bread_stock - 3
Kind: Command

Data dictionary
  Name          Type   Location
  shop          str    A1
  bread_price   float  B1
  bread_stock   int    C1
  is_open       bool   D1
  total         float  E1

Memory
  A1: Tina's Corner Shop
  B1: 2.5
  C1: 9        <- changed on this line (was 12)
  D1: True
  E1: 7.5

Screen
  Tina's Corner Shop
```

⧖ **2.10)** On sheet 6, the line `total = bread_price * 3` made a new variable. What happened to the data dictionary on that sheet that did not happen on sheet 7?

Answer. A new row was added: `total`, `float`, `E1`. On sheet 7 no row was added. `bread_stock` already had a place; only the value in `C1` changed. A new name gets a new row and a new box. An old name changes what is in its box.

Mark the box that changed on each sheet. A circle, a star, a different colour. That is how you will spot, at a glance, what each line did.

For your glossary: *address*, *data dictionary*.

Two things a program does with data

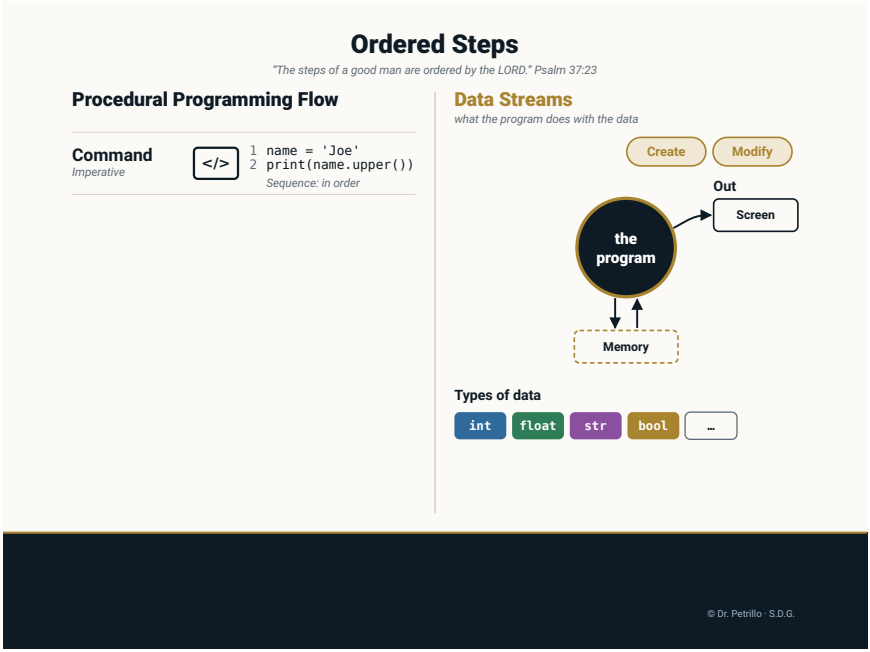
Look at lines 6 and 7 again:

- `total = bread_price * 3` **creates** a value that did not exist before: 7.5, worked out from the price.
- `bread_stock = bread_stock - 3` **modifies** a value that was already there.

Create and *modify* are two of the things a program does with data. You will meet the other two, *collect* and *move*, as the book goes on.

Your picture so far

You have added the types of data, and two of the things a program does with it: **create** and **modify**.



Exercises

1. What is in each variable at the end?

```
milk_price = 1.75
milk_stock = 20
milk_stock = milk_stock - 2
```

2. Give the type of each value: `20`, `1.75`, `"Milk"`, `False`, `"20"`, `3.0`.

3. Work out each one, then check it with Python:

```
print(10 - 4 - 3)
print(20 / 4 * 2)
print(17 // 5)
print(17 % 5)
print(2 ** 4)
```

4. Which names are allowed? Fix the ones that are not.

```
egg-price    eggPrice    egg_price    3eggs    ␣
↪while    total_sales
```

5. Find and fix the mistake:

```
egg_price = 3.00
print("Eggs cost " + egg_price)
```

6. A customer buys 2 loaves of bread, 1 carton of milk and 3 cartons of eggs. Write one line that works out the total and puts it in `total`, then print it. (It should come to 15.75.)

7. Tina's supplier delivers 150 eggs. Write a program that works out how many full cartons of 12 she can fill and how many eggs are left over, and prints both.

8. Write a program for milk like the bread program at the start of this chapter: the shop name, `milk_price`, `milk_stock`, a sale of 4 cartons, the new stock, and a line that shows `Milk left: 16`.

9. Play the memory game on paper for your program in exercise 8. Mark the box that changes on each sheet.

10. In chapter 0, exercise 8, you swapped two cups with a spare cup. Write it as a Python program: `bread_stock` starts at 20 and `milk_stock` at 12; swap them; print both. Then play the memory game on it.

Practicing with AI

> *I need help practicing Python variables, types and arithmetic. Starting simple, give me short programs with assignment statements and print that I can work out by hand before running them. Do not give me the answers. If I get stuck I'll ask for a hint. Then increase their complexity so I get better at predicting what is in memory.*

Log it as you ask: kind Practice.

Words for your glossary

Write each of these in your own words in your site's glossary: **assignment statement**, **identifier**, **type**, **integer**, **float**, **string**, **boolean**, **precedence**, **address**, **data dictionary**.

Summary

- A **variable** is a place in memory with a name. The first assignment makes it; later assignments change what is in it.
- **Assignment** `name = value`: right side first, then the answer goes in the name on the left.
- **Identifiers**: letters, digits, _; not starting with a digit; capitals matter; not Python's own words. Name things for what they hold.
- The four **base types**: `int` (whole numbers), `float` (decimals), `str` (text), `bool` (`True`/`False`). `type()` tells you which.
- **Arithmetic**: `+ - * / // % **`. `/` always gives a float. `//` is the whole part, `%` the remainder.
- **Precedence**: `*` and `/` before `+` and `-`. Brackets first. When in doubt, use brackets.

- A **float** is close, not always exact.
- `+` adds numbers and joins strings. It will not join a string to a number; use a comma in `print`.
- **Memory game**: add a memory region with addresses (`A1`, `B1`…) and a **data dictionary** (Name, Type, Location). Mark what changed.
- A program **creates** and **modifies** data.

Answers

1. `milk_price` 1.75, `milk_stock` 18.
2. `20` int; `1.75` float; `"Milk"` str; `False` bool; `"20"` str (it is in quotes); `3.0` float (it has a decimal point).
3.
```
3
10.0
3
2
16
```

   `10 - 4 - 3` is done left to right: 6, then 3. `20 / 4 * 2` is also left to right: 5.0, then 10.0.

4. Allowed: `eggPrice`, `egg_price`, `total_sales`. `egg-price` has a dash (Python reads it as *egg minus price*); use `egg_price`. `3eggs` starts with a digit; use `eggs3`. `while` is Python's own word; use something like `while_open`. Of the allowed ones, `egg_price` follows the Python habit better than `eggPrice`.
5. You cannot join a string to a float with `+`. Use a comma:

```
egg_price = 3.00
print("Eggs cost", egg_price)
```

```
Eggs cost 3.0
```

6.
```
total = 2.50 * 2 + 1.75 * 1 + 3.00 * 3
print(total)
```

```
15.75
```

7.
```
eggs = 150
cartons = eggs // 12
left_over = eggs % 12
print(cartons, left_over)
```

```
12 6
```

8.
```
shop = "Tina's Corner Shop"
milk_price = 1.75
milk_stock = 20
print(shop)
total = milk_price * 4
milk_stock = milk_stock - 4
print("Milk left:", milk_stock)
```

```
Tina's Corner Shop
Milk left: 16
```

9. Seven sheets. The changed box: sheet 1 `shop` (new), sheet 2 `milk_price` (new), sheet 3 `milk_stock` (new), sheet 4 nothing in memory (the screen changes), sheet 5 `total` (new, 7.0), sheet 6 `milk_stock` (20 to 16), sheet 7 nothing in memory (the screen changes).

10.
```
bread_stock = 20
milk_stock = 12
spare = bread_stock
bread_stock = milk_stock
milk_stock = spare
print(bread_stock, milk_stock)
```

```
12 20
```

Look at Tina's program from the last chapter. It works out what three loaves cost. Every time. Whoever walks in, whatever they want, the program sells three loaves.

That is not a shop. That is a sign with arithmetic on it.

A real customer decides. One wants two loaves, another wants five. The program has to *ask*, and then work with whatever answer comes back. In this chapter the data stops living only inside your program. It comes in from outside, from a person at the keyboard.

A program that asks a person for data and answers them is called **interactive**. Write that word in your glossary. From now on, almost every program you write will be interactive.

Two doors

Before any code, think about what happens when a program asks a question.

The program shows a message on the screen: *How many loaves?* The person reads it and types `3`. The program takes that `3` and keeps it in memory.

⧖ **3.1)** How many doors did the data pass through? Stop and answer before you read on.

Two. The question went *out* through the screen. The answer came *in* through the keyboard. They are different doors, and beginners run them together. The question is output. The answer is **input**: data that comes into the program from outside. Most of the time, for us, from the keyboard.

Write *input* in your glossary, in your own words.

input()

Python asks a question with `input()`. Here is the smallest useful example:

```
name = input("Your name: ")
print("Welcome to Tina's Corner Shop,", name)
```

When it runs, the person types `Ada` and presses Enter:

```
Your name: Ada
Welcome to Tina's Corner Shop, Ada
```

Read the first line slowly, the way the computer does. It works from the right side of the =:

1. `input("Your name: ")` shows `Your name:` on the screen and waits.
2. The person types `Ada` and presses Enter.
3. `input()` hands back what was typed, `Ada`.
4. The = puts that value in the place called `name`.

The words inside the brackets, `"Your name: "`, are called the **prompt**. The prompt is what the person sees. Notice the space after the colon. Without it, what the person types is jammed up against the prompt: `Your name:Ada`. Small things like that are the difference between a program people enjoy using and one they put up with.

⧖ **3.2)** After the first line runs, what is in memory, and where did it come from?

A place called `name` holds `Ada`. It came in through the keyboard. The program never knew the word *Ada* until the person typed it. That is the whole point.

Input is always a word

Now let us ask for the number of loaves.

```
bread_price = 2.50
count = input("How many loaves? ")
total = count * bread_price
print(total)
```

Type `3` and see what happens:

```
How many loaves? 3
Traceback (most recent call last):
  File "shop.py", line 3, in <module>
    total = count * bread_price
            ~~~~~~^~~~~~~~~~~~
TypeError: can't multiply sequence by non-int of type
↪'float'
```

⧖ **3.3)** The person typed a number. Why can't Python multiply it?

Because the person did not type a number. They typed the *character* `3`. The keyboard has no idea what a number is. It sends characters, one key at a time. So `input()` always hands back a `str`, a word, even when that word is made of digits.

Remember this sentence. You will need it many times: **input always comes in as a str.**

You can check it yourself. `type()` from the last chapter tells you what kind of value a place holds:

```
count = input("Count: ")
print(type(count))
```

```
Count: 5
<class 'str'>
```

Here is a stranger example. What do you think this prints if the person types `3`?

```
count = input("How many loaves? ")
print(count * 2)
```

⧖ **3.4)** Guess first. Then read on.

```
How many loaves? 3
33
```

Not 6. *33*. A word times 2 is the word written twice. Python was happy to

do it, and that is worse than an error, because nothing told you something was wrong. The first program crashed and you knew. This one gives a wrong answer with a straight face.

The message from the first program, `TypeError: can't multiply sequence by non-int of type 'float'`, is a good one to ask AI about. Paste the message, never your program, and log it: kind *Error*.

Turning a word into a number

To do arithmetic with what was typed, turn the word into a number. Python has a **conversion** for each base type:

You write	You get	Example
`int("7")`	a whole number	`7`
`float("2.5")`	a decimal number	`2.5`
`str(12)`	a word	`"12"`

So we put `int()` around `input()`:

```
bread_price = 2.50
count = int(input("How many loaves? "))
total = count * bread_price
print(total)
```

```
How many loaves? 3
7.5
```

Read the second line from the inside out, the way the computer does. First `input()` shows the prompt and gets the word `"3"`. Then `int()` turns the word `"3"` into the number `3`. Then = puts the number in `count`. Brackets inside brackets are worked from the inside out, the same as in mathematics.

For money, which has a decimal point, use `float()`:

```
paid = float(input("Amount paid: "))
```

⧖ **3.5)** What happens if the person types `2.5` when the program says `int(input(...))`?

```
How many loaves? 2.5
Traceback (most recent call last):
  File "shop.py", line 1, in <module>
    count = int(input("How many loaves? "))
ValueError: invalid literal for int() with base 10:
↪'2.5'
```

`int()` only turns whole numbers into numbers. `2.5` is not a whole number, so it gives up. The same thing happens if the person types `two` or presses Enter with nothing at all. You cannot stop people from typing the wrong thing, and Tina's customers will. In the chapter "When Things Go Wrong" you will learn to catch this and ask again. For now, know that it happens and why.

Showing money properly

Look at the answer above again: `7.5`. Tina's till would never show that. Money shows two places after the point: `7.50`.

Python has a neat way to put values into a line of text and say how they should look. It is called an **f-string**, because you put an `f` in front of the quotes:

```
total = 3 * 2.50
print(total)
print(f"{total:.2f}")
print(f"Total: {total:.2f}")
```

```
7.5
7.50
Total: 7.50
```

Inside an f-string, anything in curly brackets `{ }` is worked out and dropped into the text. After the colon, `.2f` means *show this as a decimal number with two places after the point*. Without a colon, the value is shown as it is: `f"Customer: {name}"` gives `Customer: Ada`.

⧖ **3.6)** Did `:.2f` change what is in `total`?

No. `total` still holds `7.5`. The `:.2f` only changes how it looks on its way out the screen door. This is an important habit of mind: *showing* a value and

changing a value are different things. When you play the memory game on this line, Memory does not change. Only the Screen does.

The receipt

Now put it together. Here is Tina's program, version 3. Type it into `shop.py` and save a copy in your site as `shop_ch03.py`.

```
# shop.py - version 3: a receipt for one customer
shop = "Tina's Corner Shop"
bread_price = 2.50
bread_stock = 12

# input
name = input("Your name: ")
count = int(input("How many loaves? "))

# process
total = count * bread_price
bread_stock = bread_stock - count

# output
print()
print(shop)
print(f"Customer: {name}")
print(f"{count} loaves at {bread_price:.2f}")
print(f"Total: {total:.2f}")
print(f"Bread left: {bread_stock}")
```

```
Your name: Ada
How many loaves? 3

Tina's Corner Shop
Customer: Ada
3 loaves at 2.50
Total: 7.50
Bread left: 9
```

The first line is a **comment**: it starts with `#`, and Python skips it. It names

the program and its version. When you have six copies of `shop.py` in your site, you will be glad each one says which it is. `print()` with nothing inside prints an empty line, so the receipt stands apart from the questions.

Notice the comments `# input`, `# process` and `# output`, and the blank lines between them. They are not decoration. They show the shape of the program, and that shape is the design tool of this chapter.

Design: input, process, output

Many programs have the same three parts:

1. **Input**: get the data.
2. **Process**: work with it, following the rules of the job.
3. **Output**: show the result.

This pattern is called **input → process → output**. Not every program fits it, but most of the small ones you will write do. When they do, split your code into those three parts, with a comment on each. It makes the program easy to read, and easier to fix, because when the output is wrong you know there are only three places to look.

Here is the part most beginners get backwards. When you design, **start with the output**. Decide what the person must see at the end. Then ask what you need to know to produce it; that is your input. The process is whatever turns one into the other.

Let us design a small program that way, before writing any code. A customer pays, and Tina needs to give change.

Output. What must the customer see?

```
Total: 7.50
Change: 2.50
```

Input. What do we need to know to work that out? The number of loaves, and how much the customer handed over. Bread's price we already know; it is not input, because nobody types it.

Process. total = count × price. change = paid − total.

Only now do we write code, and it almost writes itself:

```
bread_price = 2.50

# input
count = int(input("How many loaves? "))
paid = float(input("Amount paid: "))

# process
total = count * bread_price
change = paid - total

# output
print(f"Total: {total:.2f}")
print(f"Change: {change:.2f}")
```

```
How many loaves? 3
Amount paid: 10
Total: 7.50
Change: 2.50
```

You may wonder why bother writing the design down when the program is so short. Because the habit has to be in your hand before the programs get long. I have watched many students sit in front of a blank screen, typing and deleting, because they started with the code. Three lines on paper, *output, input, process*, would have told them where to start.

> **Helpful hint.** Write your program in pieces. Write the input part, run it, and check that the values arrived. Then write the process. Then the output. Success with a small piece gives you confidence for the next one.

The memory game: the printed template

In the last chapters you drew the memory map by hand, on blank paper. You had to decide where the screen goes, where memory goes, where to write the line you are on. Your layout was probably a little wrong. Good. That is what it was for.

Now the program has a keyboard, and there is too much to keep track of on a page you make up as you go. So here is the printed sheet, the **template**.

Download `the blank template` and print a stack of copies. One sheet for each line of code that runs.

Every region on the sheet answers one question:

Region	The question it answers
Line #	Which line of the program am I on?
Code	What does that line say? Copy it exactly.
Comment	What is this line trying to do, in plain words? Leave it blank when it is obvious.
Control Structure	Which of the six kinds of line is it? So far every line is a **Command**: do this one thing, then go to the next line.
Keyboard	What did the person type on this line?
Screen	Everything shown so far, written exactly as it appears.
Stack	For now, just the line you are on. It will start to matter in the chapter "Again, While It's True".
Data Dictionary	Every variable alive now: its **Name**, its **Type**, its **Location** (which memory cell it lives in) and its **Scope**. For now Scope is always the line it was made on, then `:G`, for *global*: it belongs to the whole program. Leave **Pass** blank.
Memory	The cells, named like a spreadsheet: `A1`, `B1`, and so on. What is in each one now.
File	Leave it blank until the chapter "Files Remember".

Do you see the cups? Memory is the egg carton, laid flat on paper with a name on each cup.

Let us play the game on this program, with Ada at the keyboard typing `3`:

```
1  bread_price = 2.50
2  name = input("Your name: ")
3  count = int(input("How many loaves? "))
4  total = count * bread_price
5  print(f"{name}: {total:.2f}")
```

Sheet for line 2.

Region	What you write
Line #	2
Code	`name = input("Your name: ")`
Control Structure	Command
Keyboard	`Ada`
Screen	`Your name: Ada`
Stack	2
Data Dictionary	`bread_price` · float · A1 · 1:G`name` · str · B1 · 2:G
Memory	A1 `2.5` · **B1 `Ada`**

Two things happened at once on this line. The prompt went out the screen door, and the answer came in the keyboard door. Write both. Then the copy went into memory at `B1`.

⧖ **3.7)** Why does `A1` say `2.5` and not `2.50`?

Because a cell holds a value, not the way it is shown. The number two and a half is the same number however you write it. Python keeps `2.5`.

Sheet for line 3.

Region	What you write
Line #	3
Code	`count = int(input("How many loaves? "))`
Control Structure	Command
Keyboard	`3`
Screen	`Your name: AdaHow many loaves? 3`
Stack	3
Data Dictionary	`bread_price` · float · A1 · 1:G`name` · str · B1 · 2:G`count` · int · C1 · 3:G
Memory	A1 `2.5` · B1 `Ada` · **C1 `3`**

The person typed the character `3`. `int()` turned it into the number `3` before it reached memory. That is why the Data Dictionary says `int`, not `str`. And

notice `B1` still holds `Ada`. Memory does not tidy itself up between lines. What you put in a cell stays there until something writes over it.

Sheet for line 4. No keyboard, no screen change. Memory gets a new cell: `D1` holds `7.5`, and the Data Dictionary gets `total` · float · D1 · 4:G.

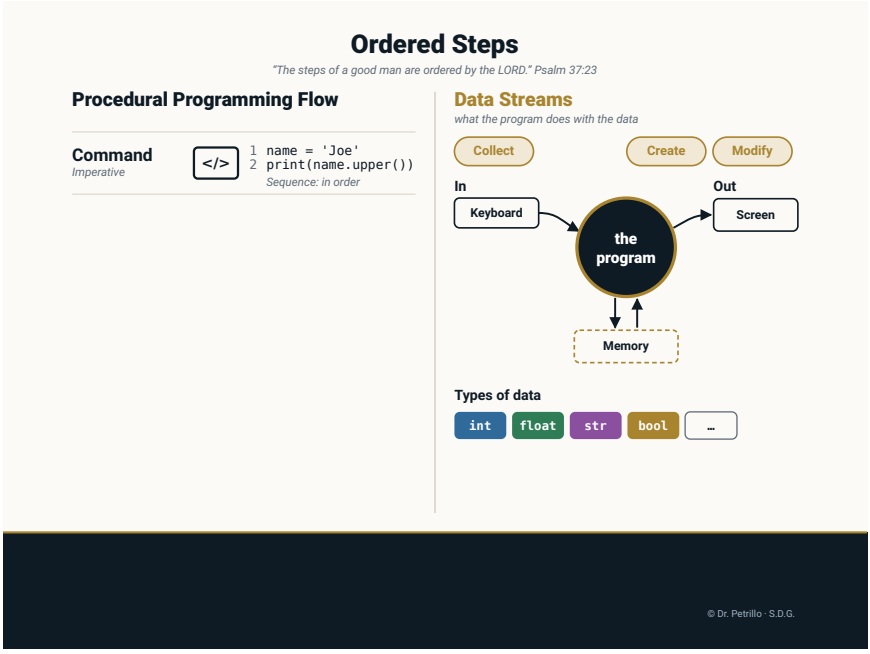
Sheet for line 5. Memory does not change at all. The Screen gets one more line: `Ada:  7.50`.

⧖ **3.8)** Line 5 changed the screen but not memory. Is that a wasted sheet?

No. It is the most useful sheet of the four. **A line that shows you something and a line that changes something are not the same line.** Students who have never drawn memory often believe `print` changed the variable, because the screen looks different afterwards. The sheet shows you it did not.

Do the game every time, even when the program seems too easy to need it. You are not tracing this program to find out what it does. You are training your eye for the day the program is too long to hold in your head.

Your picture so far



New on your picture: the **Keyboard**, the first door data comes *in* by, and **Collect**, the first thing a program does with data, gathering it from outside.

Exercises

Do these in order. They start easy and get harder. Answers are at the end of the chapter, but do each one before you look. If you cannot do it yourself, you do not understand it yet. Reading my answer does not count.

1. Without running it, write down exactly what this shows if the person types `Sam`:

```
name = input("Who is next? ")
print("Next customer:", name)
```

2. This program is meant to double the number typed. What does it print if the person types `4`? Fix it.

```
n = input("Number: ")
print(n * 2)
```

3. Write a program that asks for the customer's name and the day of the week, and prints `Good morning, Ben. Happy Tuesday!` (with their answers in it).

4. Write a receipt for milk. Ask how many bottles of milk the customer wants. Milk costs 1.75. Show the total with two places after the point.

5. What is in memory after each line, if the person types `5`? Draw the Memory region only.

```
1  stock = 12
2  sold = int(input("Sold: "))
3  stock = stock - sold
```

6. Design first, on paper: output, input, process. Then write the program. A customer wants eggs (3.00 a carton). The program asks how many cartons and how much the customer paid, and shows the total and the change.

7. Play the memory game on your answer to exercise 6, with the customer buying 2 cartons and paying 10. One template per line that runs.

8. Tina wants a receipt for all three items. Ask the customer's name, then how many loaves of bread, bottles of milk and cartons of eggs. Show one line per item with its count, price and cost, then the total.

9. A customer asks, "I have this much money. How many cartons of eggs can I buy, and what will I have left?" Design it, then write it. *Hint:* `int()` of a decimal number drops everything after the point: `int(3.9)` is `3`.

10. Find the mistake in each line, and say whether Python will stop with an error or quietly give a wrong answer:

```
a = int(input("Loaves: "))
b = input("Price: ") * a
c = float(input("Paid: ") - b)
```

Practicing with AI

When you want more practice, ask AI for it, and log it as kind *Practice*:

> *I need help practicing input in Python. Starting simple, give me programs that ask the user for data, convert it with int() or float(), and print a result. I will write them myself. If I get stuck I'll ask for a hint. Then increase their complexity so I get better at designing programs with input, process and output.*

Write every program yourself. Then play the memory game on it.

Words for your glossary

Write each of these in your own words in your site's glossary: **interactive**, **input**, **prompt**, **conversion**, **f-string**, **comment**, **input → process → output**, **template**.

Summary

- **Input** is data coming into the program from outside; for us, the keyboard. A program that asks and answers is **interactive**.

- `input("prompt")` shows the prompt, waits for Enter, and hands back what was typed.
- **Input always comes in as a `str`.** Turn it into a number with `int()` or `float()` before doing arithmetic.
- Brackets inside brackets are worked from the inside out: `int(input("Count: "))`.
- An **f-string** puts values into text: `f"Total: {total:.2f}"`. `:.2f` shows two places after the point. It changes how a value looks, not what is in memory.
- **Design: input → process → output.** Start with the output, then decide the input, then the process. Split the code into those three parts with comments.
- **The printed template** has a region for every question: Line #, Code, Comment, Control Structure, Keyboard, Screen, Stack, Data Dictionary, Memory (and File, for later). One sheet per line that runs.
- A line that shows something and a line that changes something are not the same line.

Answers

1.
```
Who is next? Sam
Next customer: Sam
```

2. It prints `44`, because `n` holds the word `"4"`, and a word times 2 is the word twice. Fix it with `n = int(input("Number: "))`, and it prints `8`.

3.
```
name = input("Your name: ")
day = input("Day of the week: ")
print(f"Good morning, {name}. Happy {day}!")
```

4.
```
milk_price = 1.75
count = int(input("How many bottles of milk? "))
total = count * milk_price
print(f"Total: {total:.2f}")
```

```
How many bottles of milk? 4
Total: 7.00
```

5. After line 1: `A1` holds `12`. After line 2: `A1` `12`, `B1` `5`. After line 3: `A1` `7`, `B1` `5`. The 12 is gone; line 3 took it out, subtracted 5, and put 7 back in the same cell.

6. Output: the total and the change. Input: the number of cartons and the amount paid. Process: total = cartons × 3.00; change = paid − total.

```
egg_price = 3.00

# input
cartons = int(input("How many cartons of eggs? "))
paid = float(input("Amount paid: "))

# process
total = cartons * egg_price
change = paid - total

# output
print(f"Total: {total:.2f}")
print(f"Change: {change:.2f}")
```

7. Five sheets, one per line that runs (the blank and comment lines do not run, so they get no sheet). Your Memory at the end: `egg_price` 3.0, `cartons` 2, `paid` 10.0, `total` 6.0, `change` 4.0. The Screen at the end:

```
How many cartons of eggs? 2
Amount paid: 10
Total: 6.00
Change: 4.00
```

   Check that your Keyboard region shows `2` on the cartons line and `10` on the paid line, and is empty on every other sheet.

8. ```
   bread_price = 2.50
   milk_price = 1.75
   ```

(continues on next page)
   ```

(continued from previous page)

```
egg_price = 3.00

# input
name = input("Your name: ")
bread = int(input("Loaves of bread: "))
milk = int(input("Bottles of milk: "))
eggs = int(input("Cartons of eggs: "))

# process
total = bread * bread_price + milk * milk_price +␣
↪eggs * egg_price

# output
print()
print("Tina's Corner Shop")
print(f"Customer: {name}")
print(f"Bread {bread} x {bread_price:.2f} =
↪{bread * bread_price:.2f}")
print(f"Milk  {milk} x {milk_price:.2f} = {milk *␣
↪milk_price:.2f}")
print(f"Eggs  {eggs} x {egg_price:.2f} = {eggs *␣
↪egg_price:.2f}")
print(f"Total: {total:.2f}")
```

```
Your name: Ben
Loaves of bread: 2
Bottles of milk: 3
Cartons of eggs: 1

Tina's Corner Shop
Customer: Ben
Bread 2 x 2.50 = 5.00
Milk  3 x 1.75 = 5.25
Eggs  1 x 3.00 = 3.00
Total: 13.25
```

Multiplication is done before addition, the same as in mathematics, so the total line needs no brackets.

9. Output: the number of cartons, and the money left. Input: the money. Process: cartons = the whole part of money ÷ 3.00; left = money − cartons × 3.00.

```
egg_price = 3.00
money = float(input("How much money do you have?
↪"))
cartons = int(money / egg_price)
left = money - cartons * egg_price
print(f"You can buy {cartons} cartons of eggs.")
print(f"You will have {left:.2f} left.")
```

```
How much money do you have? 10
You can buy 3 cartons of eggs.
You will have 1.00 left.
```

10. Line 1 is correct. Line 2 is quietly wrong: the price is never converted, so a word is repeated `a` times instead of multiplied. If the person types `2.50` and `a` is 3, `b` holds `2.502.502.50`. It should be `b = float(input("Price: ")) * a`. Line 3 stops with an error: the subtraction is inside the brackets of `float()`, so Python tries to subtract from the word that `input()` returned. It should be `c = float(input("Paid: ")) - b`.

Tina has two rules at her counter.

The first is kind: *buy ten or more of anything and you get ten percent off*. The second is plain sense: *you cannot buy fifteen loaves when there are twelve on the shelf*.

Our program from the last chapter knows neither rule. It sells ten loaves at full price, and it will happily sell fifteen loaves out of twelve and tell Tina she has `-3` left. Every line runs, every time, in order. The program cannot *choose*.

In this chapter it learns to. This is the second of the six structures: the **decision**. The program asks a question, and depending on the answer, it does one thing or another.

How would you decide?

Before any Python, think about how Tina decides.

⧖ **4.1)** A customer asks for 12 loaves. How does Tina decide whether to give the discount? Stop and answer in words before you read on.

She looks at one fact, how many loaves, and asks a question about it: *is it 10 or more?* The answer is yes or no. If yes, she takes ten percent off. If no, she doesn't. There is no third answer.

That is all a decision is. A question whose answer is yes or no, and something that depends on the answer.

True and false

You met the type `bool` already: a value that is either `True` or `False`. Now you will see where they come from. A **comparison** asks a question about two values and gives back `True` or `False`.

You write	It asks
`a == b`	Is `a` equal to `b`?
`a != b`	Is `a` not equal to `b`?
`a < b`	Is `a` less than `b`?
`a > b`	Is `a` greater than `b`?
`a <= b`	Is `a` less than or equal to `b`?
`a >= b`	Is `a` greater than or equal to `b`?

```
count = 12
print(count >= 10)
print(count == 10)
print(count != 10)
print(count < 5)
```

```
True
False
True
False
```

⧖ **4.2)** Why is it == and not =?

Because = is already taken. = means *put this value in that place*. It is an instruction, not a question. == means *are these two the same?* It is a question. Mixing them up is one of the most common mistakes there is, and Python will usually catch you:

```
    if count = 3:
       ^^^^^^^^^
SyntaxError: invalid syntax. Maybe you meant '==' or
↪':=' instead of '='?
```

Read error messages. This one even tells you the fix. (Python mentions := as well; ignore it, we will not use it.)

The whole computer rests on true and false. Every decision it will ever make comes down to a question with exactly two answers. I have always found that worth thinking about. A machine this useful works only because some things are simply true and others simply false, with nothing in between. Life needs

the same thing. If nothing is really true, there is nothing to decide by.

if

Here is the discount in Python:

```
bread_price = 2.50
count = int(input("How many loaves? "))
total = count * bread_price
if count >= 10:
    total = total * 0.9
print(f"Total: {total:.2f}")
```

```
How many loaves? 12
Total: 27.00
```

```
How many loaves? 4
Total: 10.00
```

Read the `if` line the way the computer does. It works out `count >= 10`. If the answer is `True`, it runs the lines underneath. If it is `False`, it skips them and goes on to the next line that is not pushed in.

Three things make an `if` work, and forgetting any one of them is an error:

1. The word `if`, then the question.
2. A colon `:` at the end of the line.
3. The lines that belong to the `if` are pushed in, **indented**, by four spaces.

The indented lines are called a **block**. The block is how Python knows which lines depend on the answer. The `print` line is not indented, so it does not belong to the `if`. It runs either way.

⧖ **4.3)** Why multiply by `0.9` for ten percent off?

Ten percent off means you pay ninety percent. Ninety percent of something is the something times `0.9`. Twelve loaves cost 30.00; 30.00 × 0.9 is 27.00.

if and else

The discount had one thing to do when the answer was yes, and nothing to do when it was no. The stock rule is different. It has something to do either way.

```
bread_stock = 12
count = int(input("How many loaves? "))
if count > bread_stock:
    print(f"Sorry, only {bread_stock} left.")
else:
    bread_stock = bread_stock - count
    print(f"Sold {count}. {bread_stock} left.")
```

```
How many loaves? 15
Sorry, only 12 left.
```

```
How many loaves? 5
Sold 5. 7 left.
```

`else` means *otherwise*. Exactly one of the two blocks runs, never both, never neither. Notice that `else` has its own colon, and that it lines up with `if`, not with the block.

⧖ **4.4)** When the customer asks for 15, does `bread_stock` change?

No. The line that changes it is in the `else` block, and the `else` block did not run. `bread_stock` still holds 12. That is exactly what Tina wants.

if, elif, else

Tina sells three things. The customer types a letter: `B` for bread, `M` for milk, `E` for eggs. Now there are more than two answers. Python has `elif`, short for *else if*:

```
choice = input("B)read M)ilk E)ggs: ")
if choice == "B":
    print("Bread, 2.50")
elif choice == "M":
```

(continues on next page)

(continued from previous page)

```
    print("Milk, 1.75")
elif choice == "E":
    print("Eggs, 3.00")
else:
    print("Sorry, we don't sell that.")
```

```
B)read M)ilk E)ggs: M
Milk, 1.75
```

Python asks the questions in order, top to bottom. The first one that is `True` gets its block run, and then the whole chain is finished; the rest are not even asked. If none is `True`, the `else` block runs. You may have as many `elif`s as you need, and the `else` at the end is optional.

⧖ **4.5)** What does this program print if the customer types a small `m`?

```
B)read M)ilk E)ggs: m
Sorry, we don't sell that.
```

To a computer, `m` and `M` are different characters. `"m" == "M"` is `False`. The customer meant milk, and the program turned them away.

We cannot make customers use capital letters. So we change what they typed before we compare it. A `str` has a tool for this, `.upper()`, which gives back the same word in capital letters:

```
choice = input("B)read M)ilk E)ggs: ").upper()
```

Now `m` becomes `M` before it ever reaches memory, and the program says `Milk, 1.75`. Being kind to the person at the keyboard is part of writing a good program.

and, or, not

Sometimes one question is not enough. Python joins questions with three words:

You write	It is `True` when
`a and b`	both are true
`a or b`	at least one is true
`not a`	`a` is false

```
is_open = True
choice = "M"
count = 10
if choice == "M" and count >= 10:
    print("A free bag for a big milk order!")
if choice == "Q" or choice == "X":
    print("Goodbye.")
if not is_open:
    print("Sorry, we are closed.")
```

```
A free bag for a big milk order!
```

⧖ **4.6)** With `count` holding 10, what is `count > 5 and count < 8`? What is `count > 5 or count < 8`?

`count > 5` is `True`, and `count < 8` is `False`. With `and`, both must be true, so the answer is `False`. With `or`, one is enough, so the answer is `True`.

Comparisons are worked out before `and`, `or` and `not`, so you rarely need brackets. When a line gets long, put brackets in anyway. A line that is easy to read is a line that is easy to check.

One trap. In English we say "if the choice is Q or X". In Python you must say the whole question twice: `choice == "Q" or choice == "X"`. If you write `choice == "Q" or "X"`, Python does not complain, and the answer is always true. That is the worst kind of mistake: no error, and a wrong answer.

Design: pseudocode

Now we put Tina's two rules and her menu into the shop program. That is enough pieces that it is worth planning before typing.

In the last chapter you designed with three boxes: output, input, process. Now the process has choices in it, so we write the process out in plain words, one step to a line, indented the way the code will be. That is called **pseudocode**: *pseudo* means *pretend*. It is pretend code. Nobody runs it. It is for you.

Here is the pseudocode for one customer:

```
ask which item: B, M or E
look up that item's name, price and stock
if it is not one we sell
    say sorry
otherwise
    ask how many
    if that is more than the stock
        say sorry, only so many left
    otherwise
        total = count × price
        if count is 10 or more
            take 10% off
        take the count off the stock
        print the receipt
```

Read it out loud. Does it do what Tina does? Is there a customer it gets wrong? This is the place to find mistakes, because a mistake in pseudocode costs you a pencil line. A mistake in code costs you an evening.

Notice that pseudocode has no colons, no brackets, no quotes. It is not Python. If you find yourself worrying about Python's rules while writing pseudocode, you are writing code too early.

The shop, version 4

Now each line of pseudocode becomes a few lines of Python. Save this as `shop_ch04.py` in your site.

```
# shop.py - version 4: choose an item, check the␣
↪stock, take 10% off for 10 or more
shop = "Tina's Corner Shop"
```

(continues on next page)

(continued from previous page)

```
bread_price = 2.50
milk_price = 1.75
egg_price = 3.00
bread_stock = 12
milk_stock = 20
egg_stock = 10

# input: which item
choice = input("B)read M)ilk E)ggs: ").upper()

# process: look up the item
item = ""
if choice == "B":
    item = "loaves of bread"
    price = bread_price
    stock = bread_stock
elif choice == "M":
    item = "bottles of milk"
    price = milk_price
    stock = milk_stock
elif choice == "E":
    item = "cartons of eggs"
    price = egg_price
    stock = egg_stock

if item == "":
    print("Sorry, we don't sell that.")
else:
    count = int(input(f"How many {item}? "))
    if count > stock:
        print(f"Sorry, only {stock} left.")
    else:
        total = count * price
        if count >= 10:
            total = total * 0.9
            print("10% off for 10 or more!")
        stock = stock - count
```

(continues on next page)

(continued from previous page)

```
        if choice == "B":
            bread_stock = stock
        elif choice == "M":
            milk_stock = stock
        else:
            egg_stock = stock
        # output
        print()
        print(shop)
        print(f"{count} {item} at {price:.2f}")
        print(f"Total: {total:.2f}")
        print(f"{item} left: {stock}")
```

Try it with several customers:

```
B)read M)ilk E)ggs: b
How many loaves of bread? 3

Tina's Corner Shop
3 loaves of bread at 2.50
Total: 7.50
loaves of bread left: 9
```

```
B)read M)ilk E)ggs: M
How many bottles of milk? 12
10% off for 10 or more!

Tina's Corner Shop
12 bottles of milk at 1.75
Total: 18.90
bottles of milk left: 8
```

```
B)read M)ilk E)ggs: e
How many cartons of eggs? 11
Sorry, only 10 left.
```

```
B)read M)ilk E)ggs: x
Sorry, we don't sell that.
```

A few things to notice.

`item = ""` before the menu. The empty word is a signal. If none of the menu's questions is true, `item` stays empty, and the next decision uses that to say sorry. Without it, a customer who typed `x` would reach a line that uses `item` when there is no `item` in memory, and the program would stop with an error.

An `if` inside an `if`. Decisions can sit inside the blocks of other decisions. Each level is indented four more spaces. This is called **nesting**. Follow the indents and you can see which question each line depends on. Compare it with the pseudocode: the shape is the same.

Putting the stock back. We copied the chosen item's stock into `stock`, changed `stock`, and then copied it back to the right item. In this version the program ends after one customer, so it seems pointless. In the next chapter the shop stays open, and then it matters a great deal.

You may be thinking this program is getting long, with the same kind of lines repeated for each item. You are right. Hold that thought. In the chapter "Name a Job" you will learn to give a job a name and write it once.

The memory game: a decision

A decision line is a new kind of sheet. Play it on this small program:

```
1  bread_price = 2.50
2  count = int(input("How many loaves? "))
3  total = count * bread_price
4  if count >= 10:
5      total = total * 0.9
6  print(f"Total: {total:.2f}")
```

The customer types `12`. Lines 1 to 3 are commands, as before: `A1` holds `2.5`, `B1` holds `12`, `C1` holds `30.0`.

Sheet for line 4.

Region	What you write
Line #	4
Code	`if count >= 10:`
Comment	`12 >= 10` is True, so go to line 5
Control Structure	**Decision**
Keyboard	(nothing)
Screen	`How many loaves? 12`
Stack	4
Memory	A1 `2.5` · B1 `12` · C1 `30.0`: no change

⧖ **4.7)** Look at Memory and Screen. What did line 4 produce?

Nothing you can see. Memory did not change. The Screen did not change. **A decision never writes anything. It reads memory and chooses.** The only thing it produced was a direction: which line runs next. Write that direction in the Comment, so your stack of sheets tells the story.

Sheet for line 5. `C1` changes from `30.0` to `27.0`. Control Structure: Command.

Sheet for line 6. The Screen gets `Total: 27.00`.

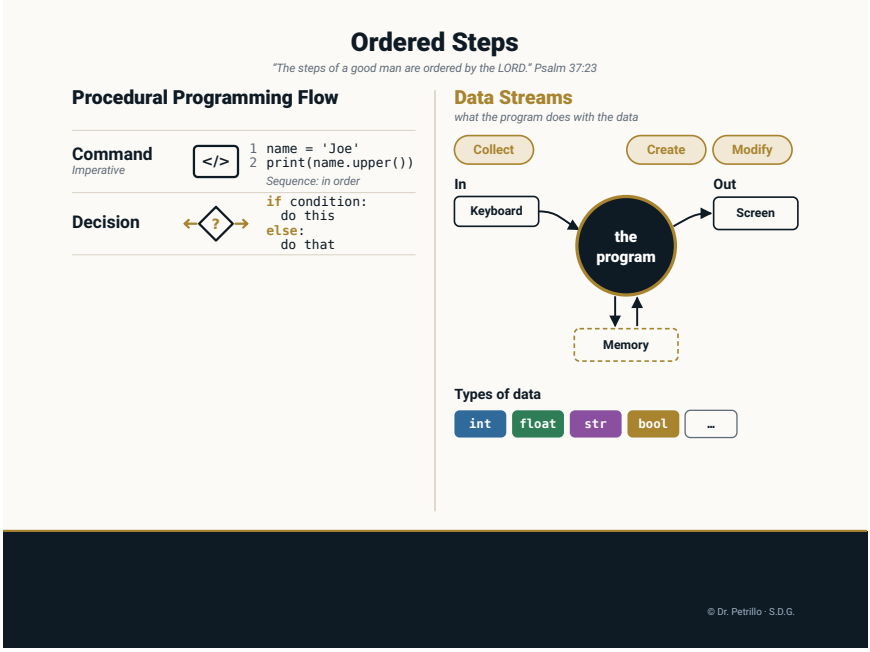
Now run it again in your head with a customer who types `4`. Line 4 reads `B1`, finds `4`, and `4 >= 10` is `False`.

⧖ **4.8)** Which sheet do you fill in for line 5?

None. Not a blank one. **None at all.** You fill in a sheet for every line that *runs*, and line 5 did not run. A student who hands me a sheet for every line of the program has not evaluated it. They copied it. With 12 at the keyboard the stack of sheets is 1, 2, 3, 4, 5, 6. With 4 it is 1, 2, 3, 4, 6. Same code, different paper, because the data was different.

That is the deepest thing a decision teaches. From now on, you cannot know which lines run without knowing what is in memory.

Your picture so far



New on your picture: **Decision**, the second of the six ways a program moves through its steps. You now have two: Command, and Decision.

Exercises

Do these in order. Answers are at the end of the chapter; do each one before you look.

1. With `count = 7` and `stock = 7`, write down what each line prints:

```
print(count < stock)
print(count <= stock)
print(count == stock and count > 5)
print(count > 10 or stock > 10)
print(not count == 7)
```

2. This program has one mistake. Find it and fix it.

```
count = int(input("Loaves: "))
if count = 0:
    print("Nothing today?")
```

3. Write a program that asks how many loaves, and prints `That's a big order!` only when it is more than 20.

4. Eggs: there are 10 cartons. Ask how many cartons the customer wants. If it is more than the stock, say `Sorry, only 10 left.` Otherwise take them off the stock and say how many are left.

5. Write the menu from this chapter so that it accepts `b`, `m` and `e` as well as `B`, `M` and `E`, and says sorry for anything else.

6. Without running it, write down what this prints for each customer: (a) `Y` and `15`; (b) `y` and `3`; (c) `N` and `3`.

```
is_open = input("Open? (Y/N) ").upper() == "Y"
count = int(input("Loaves: "))
if is_open:
    if count > 12:
        print("Not enough bread.")
    else:
        print("Coming right up.")
else:
    print("Come back tomorrow.")
print("Thank you.")
```

7. Tina gives 5% off to customers with a shop card. Write the pseudocode first, then the program: ask how many loaves and whether the customer has a card (`Y` or `N`), and show the total.

8. Play the memory game on the program in "The memory game: a decision" with a customer who types `4`. Which lines get sheets? What is on the Screen at the end?

9. Tina delivers. Delivery is free when the order is 20.00 or more *and* the customer lives 5 km away or less. Otherwise it costs 2.00. Ask for the order total and the distance, and say which.

10. Ask how many loaves and how much was paid. If it is not enough, say how much more is needed. If it is exactly right, say thank you.

Otherwise show the change. Design it with pseudocode first.

11. The shop is open from 8 until 18 (six in the evening), every day except Sunday. Ask for the day and the hour (0 to 23), and say whether the shop is open. Use `and` and `!=`. Test it with Monday at 9, Monday at 18, and Sunday at 10.

Practicing with AI

When you want more practice, ask AI, and log it as kind *Practice*:

> *I need help practicing if statements. In other words, starting simple, give me programs that use if statements that I can write. If I get stuck I'll ask for a hint. Then increase their complexity so I get better at designing programs with decisions.*

Write every program yourself, and play the memory game on it. Count your sheets: they should not match the number of lines.

Words for your glossary

Write each of these in your own words in your site's glossary: **decision**, **comparison**, `bool`, **block**, **indent**, `elif`, `and`, `or`, `not`, **nesting**, **pseudocode**.

Summary

- A **decision** asks a question whose answer is `True` or `False`, and runs different lines depending on the answer.
- **Comparisons** give `True` or `False`: `==`, `!=`, `<`, `>`, `<=`, `>=`. `=` puts a value in a place; `==` asks whether two values are equal.
- `if question:` then an indented **block**. The block runs only when the answer is `True`.
- `else:` runs when the answer is `False`. `elif question:` asks the next question. In a chain, only the first true block runs.
- `and` (both), `or` (at least one), `not` (the opposite). Write the whole question each time: `choice == "Q" or choice == "X"`.

- `.upper()` gives back a word in capital letters, so `m` and `M` can be treated alike.
- Decisions can be **nested**, one inside another; the indents show what depends on what.
- **Design: pseudocode.** The process in plain words, one step per line, indented like the code will be. Find the mistakes there.
- **The memory game:** a decision line changes neither Memory nor Screen; write its direction in the Comment. A line that does not run gets no sheet.

Answers

1.
```
False
True
True
False
False
```

2. `if count = 0:` uses =, which puts a value in a place. The question needs ==: `if count == 0:`.

3.
```
count = int(input("How many loaves? "))
if count > 20:
    print("That's a big order!")
```

4.
```
egg_stock = 10
cartons = int(input("How many cartons of eggs? "))
if cartons > egg_stock:
    print(f"Sorry, only {egg_stock} left.")
else:
    egg_stock = egg_stock - cartons
    print(f"Done. {egg_stock} cartons left.")
```

```
How many cartons of eggs? 4
Done. 6 cartons left.
```

5. Add `.upper()` to the input line; the rest stays the same.

```
choice = input("B)read M)ilk E)ggs: ").upper()
if choice == "B":
    print("Bread, 2.50")
elif choice == "M":
    print("Milk, 1.75")
elif choice == "E":
    print("Eggs, 3.00")
else:
    print("Sorry, we don't sell that.")
```

6. (a) `Not enough bread.` then `Thank you.` (b) `Coming right up.` then `Thank you.` (c) `Come back tomorrow.` then `Thank you.` The last line is not indented, so it runs every time. In (b), the small `y` becomes `Y` before the comparison, so `is_open` holds `True`.

7.
```
ask how many loaves
ask if they have a card
total = count × 2.50
if they have a card
    take 5% off
show the total
```

```
bread_price = 2.50
count = int(input("How many loaves? "))
card = input("Do you have a shop card? (Y/N) ").
↪upper()
total = count * bread_price
if card == "Y":
    total = total * 0.95
print(f"Total: {total:.2f}")
```

```
How many loaves? 4
Do you have a shop card? (Y/N) y
Total: 9.50
```

8. Lines 1, 2, 3, 4 and 6 get sheets. Line 5 gets none, because `4 >= 10` is `False`. On line 4 the Comment says `4 >= 10 is False, so go to line 6`. At the end, Memory holds `A1 2.5`, `B1 4`, `C1 10.0`,

and the Screen shows:

```
How many loaves? 4
Total: 10.00
```

9.
```
total = float(input("Order total: "))
distance = float(input("Distance in km: "))
if total >= 20 and distance <= 5:
    print("Free delivery!")
else:
    print("Delivery costs 2.00.")
```

10.
```
ask how many loaves and how much was paid
total = count × 2.50
if paid is less than total
    say how much more is needed
otherwise, if paid is exactly the total
    say thank you
otherwise
    show the change
```

```
bread_price = 2.50
count = int(input("How many loaves? "))
paid = float(input("Amount paid: "))
total = count * bread_price
if paid < total:
    print(f"Not enough. You need {total - paid:.
↪2f} more.")
elif paid == total:
    print("Exactly right. Thank you!")
else:
    print(f"Change: {paid - total:.2f}")
```

```
How many loaves? 3
Amount paid: 5
Not enough. You need 2.50 more.
```

11.
```
day = input("Day: ")
hour = int(input("Hour (0-23): "))
if day != "Sunday" and hour >= 8 and hour < 18:
    print("Tina's Corner Shop is open.")
else:
    print("Sorry, we are closed.")
```

Monday at 9: open. Monday at 18: closed, because `18 < 18` is `False`; the shop closes *at* six. Sunday at 10: closed. Testing the edge, the exact hour the shop closes, is how you catch the difference between < and <=.

Tina's program serves one customer and stops. Then she has to run it again for the next customer, and the next. By the end of the day the program has forgotten every sale but the last. It cannot tell her how much she took in, because each run starts fresh.

A shop does not open for one customer. It opens in the morning and keeps selling until closing time. Our program needs to do the same thing: go round again, and again, as long as the shop is open.

That is the third of the six structures, the **loop**: go round again while something stays true.

Sam sweeps the shop

Here is a story I have told students for thirty years, with a new shop in it.

Tina's nephew Sam comes to stay for the school holiday, four weeks. Tina asks him to sweep the shop on weekdays, and offers to pay him.

"How much do you want?" she asks.

"One coin the first day," says Sam, "two coins the second day, four the third, and keep doubling until I go home."

Tina laughs. *His mother says he is good at mathematics, but he doesn't understand money.* "One coin is nothing these days," she says. "Wouldn't you rather have a hundred coins a day?"

"No, thank you," says Sam. "I'd rather have my coins doubled."

Sam was to sweep for twenty days. Tina stopped him after ten. Why?

⧖ **5.1)** You already know enough Python to find out. How would you write a program that shows Sam's pay for each of the twenty days? Think about it before you read on.

With what you know, you would write something like this:

```
pay = 1
print(f"Day 1 pay: {pay}")
pay = pay + pay
print(f"Day 2 pay: {pay}")
pay = pay + pay
print(f"Day 3 pay: {pay}")
pay = pay + pay
```

and so on, forty lines, until day 20. It works. It is also more work than doing the sums on paper, and if Sam stayed a year you would need seven hundred lines.

Look at what those lines do. The same two lines, *print the pay, double the pay*, over and over. The only thing that changes is the day number. When you hear yourself say *and so on, until*, you need a loop.

while

Here is Sam's pay with a loop:

```
# pay.py - Sam's pay, doubled every day he sweeps
pay = 1
day = 1
while day <= 20:
    print(f"Day {day} pay: {pay}")
    pay = pay + pay
    day = day + 1
print("End of program.")
```

```
Day 1 pay: 1
Day 2 pay: 2
Day 3 pay: 4
...
Day 9 pay: 256
Day 10 pay: 512
Day 11 pay: 1024
...
Day 19 pay: 262144
```

(continues on next page)

(continued from previous page)

```
Day 20 pay: 524288
End of program.
```

(The dots stand for lines I have left out to save space. Your screen shows all twenty.)

Now you see why Tina stopped him. By day 11 she owed him more than a thousand coins in a single day. On day 20 she would owe him over half a million. Sam was not bad at money. I wish he were my banker.

A `while` looks like an `if`. It has a question, a colon, and an indented block. The difference is what happens after the block:

- An `if` asks once. If the answer is true, it runs the block, and then it goes on down the program.
- A `while` asks, runs the block, and then **goes back and asks again**. It keeps going round, *test, block, test, block*, until the answer is `False`. Then it goes on to the first line after the block.

⧖ **5.2)** How many times does the block run? How many times is the question `day <= 20` asked?

The block runs 20 times, once for each day from 1 to 20. The question is asked **21** times. The 21st time, `day` holds 21, `21 <= 20` is `False`, and the loop ends. The question is always asked one more time than the block runs, because something has to say *stop*.

Each time round a loop is called an **iteration**. Write that in your glossary; it is the other name for this structure.

Three kinds of variable a loop needs

Loops use variables in a few common ways. Each has a name, and each goes in your glossary.

A counter counts how many times something happened. It starts at 0 or 1, and goes up by one: `day = day + 1`. Here is the smallest loop with a counter:

```
day = 1
while day <= 5:
    print("Day", day)
    day = day + 1
print("Done")
```

```
Day 1
Day 2
Day 3
Day 4
Day 5
Done
```

A running total, also called an **accumulator**, adds something up as the loop goes. It starts at 0, and each time round the loop something is added to it: `day_total = day_total + total`. At the end of the day, Tina's running total is everything she took in.

A sentinel is a special value that means *stop*. A guard at a gate is a sentinel; he stands there until the signal comes. Tina's program will use `Q` for quit. Here is a loop that adds up sales until Tina types `0`:

```
day_total = 0.0
sales = 0
amount = float(input("Sale amount (0 to stop): "))
while amount != 0:
    day_total = day_total + amount
    sales = sales + 1
    amount = float(input("Sale amount (0 to stop): "))
print(f"{sales} sales, {day_total:.2f} in all")
```

```
Sale amount (0 to stop): 7.50
Sale amount (0 to stop): 3.50
Sale amount (0 to stop): 18.90
Sale amount (0 to stop): 0
3 sales, 29.90 in all
```

This program uses all three. `sales` is a counter, `day_total` is a running total, and `0` is the sentinel.

⧖ **5.3)** Why does the program ask for an amount in two places, once before the loop and once at the bottom of the block?

Because the question `amount != 0` needs an amount to look at the very first time it is asked. The first `input` fills `amount` before the loop starts. The second one, at the bottom of the block, gets the next amount just before the question is asked again. Take away the first one and the loop has nothing to test. Take away the second and `amount` never changes, which brings us to the loop that never ends.

The loop that never ends

```
egg_stock = 3
while egg_stock > 0:
    print("Carton sold.")
```

⧖ **5.4)** What does this program do?

It prints `Carton sold.` forever. Nothing in the block changes `egg_stock`, so `3 > 0` is `True` every time it is asked. This is an **infinite loop**. To stop it, hold down Ctrl and press C. Python stops with `KeyboardInterrupt`.

Every `while` needs something in its block that can make the question false. Before you run a loop, put your finger on that line. If you cannot find it, you have an infinite loop.

The fix here is one line, `egg_stock = egg_stock - 1`, inside the block.

The memory game: the Stack begins

Until now the Stack on your sheet has held one number, the line you are on. Now it starts doing its real job.

Play the game on this program:

```
1  sold = 0
2  total = 0.0
3  while sold < 3:
4      total = total + 2.50
```

(continues on next page)

(continued from previous page)

```
5       sold = sold + 1
6   print(f"{sold} loaves, {total:.2f}")
```

Here is the rule for the Stack. It fits in three words: **stack and back**. The top of the Stack is the line you are on. Anything underneath is a line you must **come back to**.

Line 3 is a line the flow must come back to. After line 5, the program does not fall down to line 6. It goes back up to line 3 to ask the question again. So while the block is running, `3` waits on the Stack, underneath the line you are on.[1]

Here are all the sheets, shortened to the regions that change. Write the Stack bottom first: `3, 4` means `4` is on top, sitting on the `3` you owe.

[1] This is a teaching choice, not a description of the machine. A computer runs a loop by jumping back to the loop's question when the block finishes; nothing is added to its real call stack. The memory game puts the loop's first line on the Stack anyway, so that you practise the idea of a line waiting to be returned to before functions need it.

Shee	Line	Control Structure	Stacl	`sold` (A1)	`total` (B1)	Comment
1	1	Command	1	0		
2	2	Command	2	0	0.0	
3	3	Loop	3	0	0.0	`0 < 3` True, go to 4
4	4	Command	3, 4	0	**2.5**	
5	5	Command	3, 5	**1**	2.5	back to 3
6	3	Loop	3	1	2.5	`1 < 3` True, go to 4
7	4	Command	3, 4	1	**5.0**	
8	5	Command	3, 5	**2**	5.0	back to 3
9	3	Loop	3	2	5.0	`2 < 3` True, go to 4
10	4	Command	3, 4	2	**7.5**	
11	5	Command	3, 5	**3**	7.5	back to 3
12	3	Loop	3	3	7.5	`3 < 3` False, go to 6
13	6	Command	6	3	7.5	Screen: `3 loaves, 7.50`

⧖ **5.5)** The program has six lines. How many sheets did it take?

Thirteen. This is the first time the number of sheets does not match the number of lines, and that is the whole idea of a loop, made visible. Line 3 got four sheets. Lines 4 and 5 got three each. Line 6 got one. **The paper count is the program's real length.** What is written and what is run are different things, and now you have felt the difference in your hand.

Watch the Stack column. Every sheet inside the block shows two numbers: the line you are on, and the `3` waiting underneath. When the question finally comes back `False`, nothing is owed any more. The `3` goes, and line 6 stands alone.

⧖ **5.6)** How would the sheets show you an infinite loop?

You would see the same Stack, `3`, then `3, 4`, then `3`, over and over, and the cell the question reads would never change. An infinite loop is almost always a question reading a cell that nothing in the block writes. The sheets point

straight at it.

Look closely at sheets 4 and 5. `total = total + 2.50` is not an equation. Read as mathematics, it is nonsense; nothing equals itself plus two and a half. Read as the computer reads it, the right side is worked out first from what is in the cell *now*, and the answer is put back in the same cell. You did exactly this with the egg carton: take the value out of the cup, add to it, put it back.

You may be wondering why we watch the Stack so closely for a loop. You will see in the chapter "Name a Job". A function works the same way, and when you get there, there will be almost nothing new to learn.

The memory game moves to the spreadsheet

Thirteen sheets for six lines. Tina's shop program, with a few customers, would take a hundred. You will feel the work of it, especially the copying. Every sheet repeats almost everything from the sheet before, with one or two cells changed.

That is the moment for the next stage of the game: the **spreadsheet**. A spreadsheet already *is* a grid of labelled cells, which is exactly what Memory is. That is why the Memory region on your template is lettered like a spreadsheet: `A1`, `B1`, `C1`, across, and rows 1, 2, 3, down.

Lay one sheet out as a block of rows:

Line #	5
Code	`sold = sold + 1`
Comment	back to 3
Control Structure	Command
Keyboard	
Screen	
Stack	3, 5

Name	Type	Location	Scope
`sold`	int	A1	1:G
`total`	float	B1	2:G

Memory

	A	B	C	D	E	F	G	H
1	1	2.5						
2								
3								

Then:

1. Copy the whole block down, below itself, for the next line that runs.
2. Change only what that line changed. Leave everything else alone.
3. Colour the cells that changed, so your eye finds them.

That copying is the point. The cells you do *not* touch are as much a part of the answer as the ones you do. The spreadsheet removes the labour of writing everything again; it does not remove the thinking. You still decide, line by line, what changed. One block of rows per sheet on one tab works, or one tab per sheet. Never mix two sheets in one block.

Design: stepwise refinement

Now we turn Tina's program into a shop that stays open. It is the longest program you have written, and this is where many students sink. They type the whole thing, run it, get six errors, fix three, make two more, and give up.

There is a better way, and it is the design tool of this chapter: **stepwise refinement**. Write a little. Run it. Test it. Play the memory game on the part you are not sure of. Only when that little piece works, add the next little piece. Each step makes the program a bit more refined, a bit closer to what Tina needs.

Years ago my students built a menu program for a drinks seller this way, each step tested alone before the next was written. Every one of them finished it. Here is the same thing for Tina.

Step 1: a loop that stops on Q. Nothing else.

```
choice = ""
while choice != "Q":
    choice = input("B)read M)ilk E)ggs Q)uit: ").
↪upper()
print("Tina's Corner Shop is closed.")
```

```
B)read M)ilk E)ggs Q)uit: b
B)read M)ilk E)ggs Q)uit: m
B)read M)ilk E)ggs Q)uit: q
Tina's Corner Shop is closed.
```

It does nothing useful yet. That is fine. It proves the loop goes round and that `q` stops it. Why does `choice = ""` come first? Because the question `choice != "Q"` needs something in `choice` the first time it is asked. The empty word is not `Q`, so the loop starts.

Step 2: the menu. Inside the loop, look up the item, as in the last chapter. Test that each letter, and a wrong letter, does what it should.

Step 3: the count and the total.

```
bread_price = 2.50
milk_price = 1.75
egg_price = 3.00

choice = ""
while choice != "Q":
    choice = input("B)read M)ilk E)ggs Q)uit: ").
↪upper()
    item = ""
    if choice == "B":
        item = "loaves of bread"
        price = bread_price
    elif choice == "M":
        item = "bottles of milk"
        price = milk_price
    elif choice == "E":
        item = "cartons of eggs"
        price = egg_price
```

(continues on next page)

(continued from previous page)

```
    if choice == "Q":
        print("Closing time.")
    elif item == "":
        print("Sorry, we don't sell that.")
    else:
        count = int(input(f"How many {item}? "))
        total = count * price
        print(f"Total: {total:.2f}")
```

```
B)read M)ilk E)ggs Q)uit: m
How many bottles of milk? 2
Total: 3.50
B)read M)ilk E)ggs Q)uit: b
How many loaves of bread? 1
Total: 2.50
B)read M)ilk E)ggs Q)uit: q
Closing time.
```

⧖ **5.7)** Why the new `if choice == "Q":` branch?

Because without it, typing `Q` would find `item` empty and say *Sorry, we don't sell that* on the way out. Testing step 3 is how you find that. A small step, tested, shows you its own mistakes while they are still small.

Step 4: the stock and the discount, brought in from the last chapter. Now it matters that each item's stock is put back after a sale, because the next customer is in the same run.

Step 5: the day's totals. A counter for the number of sales, a running total for the money, and a summary after the loop.

Here is the result. Save it as `shop_ch05.py` in your site.

```
# shop.py - version 5: keep selling until Tina closes␣
↪the shop
shop = "Tina's Corner Shop"
bread_price = 2.50
milk_price = 1.75
```

(continues on next page)

(continued from previous page)

```
egg_price = 3.00
bread_stock = 12
milk_stock = 20
egg_stock = 10
day_total = 0.0
sales = 0

print(shop)
choice = ""
while choice != "Q":
    choice = input("B)read M)ilk E)ggs Q)uit: ").
↪upper()

    # look up the item
    item = ""
    if choice == "B":
        item = "loaves of bread"
        price = bread_price
        stock = bread_stock
    elif choice == "M":
        item = "bottles of milk"
        price = milk_price
        stock = milk_stock
    elif choice == "E":
        item = "cartons of eggs"
        price = egg_price
        stock = egg_stock

    if choice == "Q":
        print("Closing time.")
    elif item == "":
        print("Sorry, we don't sell that.")
    else:
        count = int(input(f"How many {item}? "))
        if count > stock:
            print(f"Sorry, only {stock} left.")
        else:
```

(continues on next page)

(continued from previous page)

```
            total = count * price
            if count >= 10:
                total = total * 0.9
                print("10% off for 10 or more!")
            stock = stock - count
            if choice == "B":
                bread_stock = stock
            elif choice == "M":
                milk_stock = stock
            else:
                egg_stock = stock
            day_total = day_total + total
            sales = sales + 1
            print(f"Total: {total:.2f}")

# the day's summary
print()
print(f"Sales today: {sales}")
print(f"Taken in: {day_total:.2f}")
print(f"Left: bread {bread_stock}, milk {milk_stock},␣
↪eggs {egg_stock}")
```

A day at the shop:

```
Tina's Corner Shop
B)read M)ilk E)ggs Q)uit: b
How many loaves of bread? 3
Total: 7.50
B)read M)ilk E)ggs Q)uit: m
How many bottles of milk? 12
10% off for 10 or more!
Total: 18.90
B)read M)ilk E)ggs Q)uit: e
How many cartons of eggs? 11
Sorry, only 10 left.
B)read M)ilk E)ggs Q)uit: x
Sorry, we don't sell that.
```

(continues on next page)

(continued from previous page)

```
B)read M)ilk E)ggs Q)uit: b
How many loaves of bread? 4
Total: 10.00
B)read M)ilk E)ggs Q)uit: q
Closing time.

Sales today: 3
Taken in: 36.40
Left: bread 5, milk 8, eggs 10
```

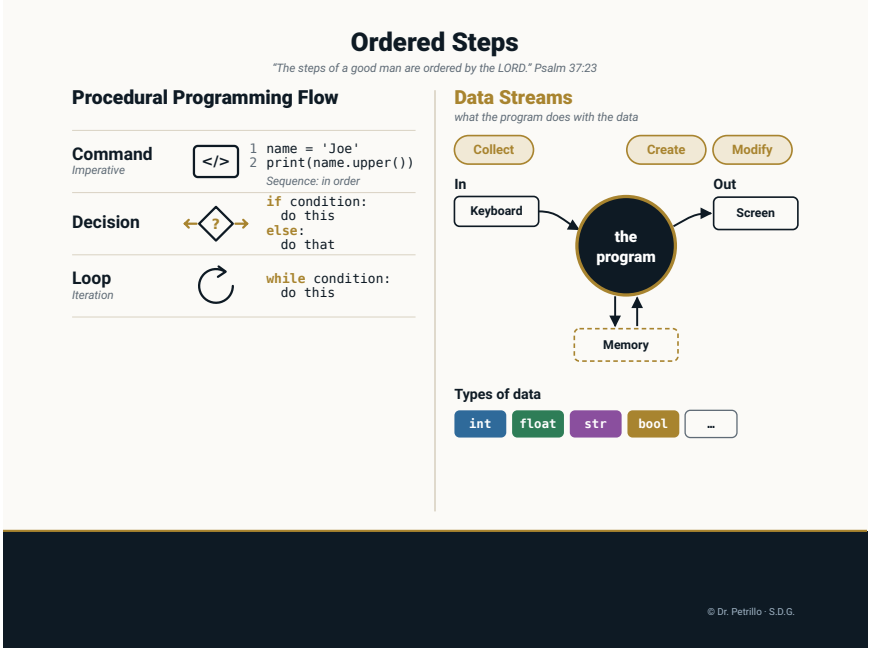
⧖ **5.8)** The second bread customer saw the stock as 9, not 12. Which lines made that happen?

`stock = stock - count` took the 3 loaves off `stock`, and then `bread_stock = stock` put the new number back in `bread_stock`. The next time round the loop, `stock = bread_stock` picked up 9. Without the put-back line, every customer would see 12 loaves. That line looked pointless in the last chapter; in a loop, it is what makes the shop remember.

⧖ **5.9)** What happens if a customer types `two` when asked how many?

The program stops with `ValueError`, and Tina loses the whole day's totals. That is serious, and it is exactly what the next chapter fixes.

Your picture so far



New on your picture: **Loop**, the third of the six. You have Command, Decision and Loop now, and with them you can write a program that runs all day.

Exercises

Do these in order; answers are at the end. Play the memory game on any loop you are not sure of. It will show you.

1. Without running it, write down what this prints:

```
n = 7
while n > 0:
    print(n)
    n = n - 2
print("Done")
```

2. This loop never ends. Why? Fix it so it prints the stock going down from 3 to 1.

```
stock = 3
while stock > 0:
    print("Stock:", stock)
```

3. Write a loop that prints the numbers 1 to 10, one per line.
4. Ask for customer names until Tina types `done`. Greet each one with `Welcome, Ada!`, and at the end say how many customers there were. Which variable is your counter, and what is your sentinel?
5. Tina has 3 cartons of eggs. Write a loop that sells one carton each time round, printing `Carton sold. 2 left.` and so on, and prints `Sold out!` when there are none.
6. Play the memory game on your answer to exercise 5, as a table like the one in this chapter, with the Stack column. How many sheets?
7. Ask for menu letters (`B`, `M`, `E`, or `Q` to quit) and count how many times each item was chosen. Say sorry for any other letter. At the end, print something like `Bread 2, milk 1, eggs 0`.
8. Tina stopped Sam after 10 days. How much did she pay him in all? Change the pay program so it adds up the pay with a running total and prints the total for the 10 days.
9. Ask `Do you want a bag? (Y/N)`. If the customer types anything other than `Y` or `N` (small letters are fine), say `Please type Y or N.` and ask again, until they do.
10. Take the sale-amount program from this chapter (the one that stops on `0`) and add the average sale at the end. Be careful: what happens if the very first amount is `0`? Make the program say `No sales today.` instead of crashing.
11. On which day does Sam's pay first go over 1000 coins? Write a loop that keeps doubling until it does, and then prints the day and the pay. Hint: this time the question is about `pay`, not `day`.

Practicing with AI

For more practice, ask AI, and log it as kind *Practice*:

> *I need help practicing while loops. Starting simple, give me programs that use while loops with counters, running totals and sentinel values that I can write. If I get stuck I'll ask for a hint. Then increase their complexity so I get better at designing programs that repeat.*

Write every program yourself, and play the memory game on it in your spreadsheet. Watch the Stack.

Words for your glossary

Write each of these in your own words in your site's glossary: **loop**, `while`, **iteration**, **counter**, **running total (accumulator)**, **sentinel**, **infinite loop**, **Stack**, **stepwise refinement**.

Summary

- A **loop** goes round again while something stays true. Each time round is an **iteration**.
- `while question:` then an indented block. It asks, runs the block, and goes back to ask again, until the answer is `False`. The question is asked one more time than the block runs.
- A **counter** counts (`n = n + 1`). A **running total**, or **accumulator**, adds up (`total = total + amount`). A **sentinel** is a value that means stop (`Q`, `0`, `done`).
- Give the loop's question something to look at before the loop starts, and change it inside the block.
- An **infinite loop** never makes its question false. Ctrl+C stops it. Find the line in the block that can make the question false; if there is none, that is the bug.
- **Stack and back:** the top of the Stack is the line you are on; underneath is a line you must come back to. Inside a loop, the loop's first line waits underneath.

- The number of sheets is the program's real length. A loop is where it first differs from the number of lines.
- **The spreadsheet** is the next stage of the memory game: one block of rows per sheet, copied down, changing only what the line changed.
- **Design: stepwise refinement.** Write a little, run it, test it, then add the next little piece.

Answers

1. ```
   7
   5
   3
   1
   Done
   ```

   After printing 1, `n` becomes `-1`, and `-1 > 0` is `False`.

2. Nothing in the block changes `stock`, so `stock > 0` is always `True`. Add `stock = stock - 1` inside the block:

   ```
   stock = 3
   while stock > 0:
       print("Stock:", stock)
       stock = stock - 1
   ```

3. ```
   n = 1
   while n <= 10:
       print(n)
       n = n + 1
   ```

4. `customers` is the counter; `done` is the sentinel.

   ```
   customers = 0
   name = input("Customer name (done to stop): ")
   while name != "done":
       customers = customers + 1
       print(f"Welcome, {name}!")
   ```

(continues on next page)

(continued from previous page)

```
    name = input("Customer name (done to stop): ")
print(f"{customers} customers today.")
```

```
Customer name (done to stop): Ada
Welcome, Ada!
Customer name (done to stop): Ben
Welcome, Ben!
Customer name (done to stop): Cy
Welcome, Cy!
Customer name (done to stop): done
3 customers today.
```

5.
```
egg_stock = 3
while egg_stock > 0:
    egg_stock = egg_stock - 1
    print(f"Carton sold. {egg_stock} left.")
print("Sold out!")
```

```
Carton sold. 2 left.
Carton sold. 1 left.
Carton sold. 0 left.
Sold out!
```

6. Numbering the lines 1 to 5, the sheets are lines 1, 2, 3, 4, 2, 3, 4, 2, 3, 4, 2, 5: twelve sheets. The question on line 2 is asked four times; the fourth time `0 > 0` is `False`. On every sheet for lines 3 and 4 the Stack reads `2, 3` or `2, 4`. On the last sheet it reads `5`.

7.
```
bread = 0
milk = 0
eggs = 0
choice = ""
while choice != "Q":
    choice = input("B)read M)ilk E)ggs Q)uit: ").
↪upper()
    if choice == "B":
        bread = bread + 1
```

(continues on next page)

(continued from previous page)

```
    elif choice == "M":
        milk = milk + 1
    elif choice == "E":
        eggs = eggs + 1
    elif choice != "Q":
        print("Sorry, we don't sell that.")
print(f"Bread {bread}, milk {milk}, eggs {eggs}")
```

With `b`, `m`, `b`, `z`, `q`, it prints `Sorry, we don't sell that.` once, then `Bread 2, milk 1, eggs 0`.

8. 1023 coins.

```
pay = 1
day = 1
paid = 0
while day <= 10:
    paid = paid + pay
    pay = pay + pay
    day = day + 1
print(f"Total paid for 10 days: {paid}")
```

9.
```
answer = input("Do you want a bag? (Y/N) ").
↪upper()
while answer != "Y" and answer != "N":
    print("Please type Y or N.")
    answer = input("Do you want a bag? (Y/N) ").
↪upper()
print(f"You said {answer}.")
```

Notice the `and`. The loop must go round while the answer is not `Y` *and* not `N`. With `or`, it would never stop, because every answer is either not `Y` or not `N`.

10. If the first amount is `0`, `sales` is 0, and dividing by 0 stops the program with `ZeroDivisionError`. A decision guards it:

```
day_total = 0.0
```

(continues on next page)

(continued from previous page)

```
sales = 0
amount = float(input("Sale amount (0 to stop): "))
while amount != 0:
    day_total = day_total + amount
    sales = sales + 1
    amount = float(input("Sale amount (0 to␣
↪stop): "))
print(f"{sales} sales, {day_total:.2f} in all")
if sales > 0:
    print(f"Average sale: {day_total / sales:.2f}
↪")
else:
    print("No sales today.")
```

With 7.50, 3.50, 18.90 and 0, the average is `9.97`.

11. Day 11, with 1024 coins.

```
pay = 1
day = 1
while pay <= 1000:
    pay = pay + pay
    day = day + 1
print(f"On day {day} the pay is {pay}.")
```

A customer types “two”

Tina’s shop program works. You built it in the last chapter: it shows the menu, asks how many, checks the stock, gives a discount for ten or more, and keeps selling until someone presses `Q`.

Then a customer comes in. The program asks *How many loaves of bread?* and the customer, being a person and not a computer, types the word `two`.

The program stops dead:

```
B)read M)ilk E)ggs Q)uit: b
How many loaves of bread? two
Traceback (most recent call last):
  File "shop.py", line 37, in <module>
    count = int(input(f"How many {item}? "))
ValueError: invalid literal for int() with base 10:
↪'two'
```

The shop is closed. Every sale in `day_total` is gone, because the program ended before it could print the day’s summary.

Whose fault is it? You may want to say the customer’s. But a customer is supposed to be a customer, not a programmer. People type `two`. They type a lowercase `l` when they mean the number `1`; when I was young, typists did that all the time, because old typewriters had no `1` key. People press Enter before typing anything. A program that only works when people do everything right does not work.

This chapter is about the fifth of the six structures on the banner: the **exception**.

Three kinds of errors

Before we fix the shop, let's name what can go wrong. There are three kinds of errors, and they show up at different times.

Kind	When you find out	Example
Syntax error	Before the program runs at all	`print("Hello"` — the closing `)` is missing
Run-time error	While the program is running, when it reaches the bad line	`int("two")`
Logic error	Maybe never. The program runs and gives the wrong answer	`total = count + price` when you meant `*`

> **For your glossary:** *syntax error*, *runtime error*, *logic error*. Put all three in your glossary, in your own words.

A *syntax error* is like a sentence with the full stop in the wrong place. Python reads the whole program first, and if it cannot understand the grammar, it will not run any of it:

```
  File "hello.py", line 1
    print("Hello"
          ^
SyntaxError: '(' was never closed
```

A *runtime error* happens in a program that is grammatically fine. Python started running it, got to a line, and could not do what the line said. `int("two")` is perfectly good Python grammar. It just cannot be done: there is no whole number called `two` as far as `int()` knows.

A *logic error* is the most dangerous of the three, because nothing tells you. The computer did exactly what you said. What you said was wrong. The memory game is how you catch these: you walk the program line by line and see where the value in a cup stops being what you expected.

⧖ **6.1)** Which kind of error is each of these?

```
total = count * price          # (a) you meant to add␣
```

(continues on next page)

(continued from previous page)

```
↪the 10% discount but forgot
print("Total:" total)           # (b)
count = int("12 loaves")        # (c)
```

Answer: (a) is a logic error: it runs, the answer is wrong. (b) is a syntax error: a comma is missing between `"Total:"` and `total`, so Python cannot read the line. (c) is a runtime error: the grammar is fine, but `"12 loaves"` is not a whole number, so `int()` fails when that line runs.

Reading a traceback

When a runtime error happens, Python prints a *traceback*. Beginners see a wall of red and panic. Don't. A traceback is the computer telling you exactly where it got stuck. Read it **from the bottom up**.

> **For your glossary:** *traceback* — the report Python prints when a runtime error stops the program.

```
Traceback (most recent call last):
  File "shop.py", line 37, in <module>
    count = int(input(f"How many {item}? "))
ValueError: invalid literal for int() with base 10:
↪'two'
```

1. **The last line** says *what* went wrong. The word before the colon is the kind of error: `ValueError`. After the colon is the message: `int()` was given `'two'`, and that is not a whole number written in base 10, our ordinary counting numbers.
2. **The line above it** is the line of code that failed, copied out for you.
3. **The line above that** says *where*: the file `shop.py`, line 37.

So, from the bottom: *a ValueError, because `'two'` is not a whole number, on line 37 of shop.py.* That is everything you need.

⧖ **6.2)** What would the last line of the traceback say if the customer typed `2.5`?

Answer: `ValueError: invalid literal for int() with base 10: '2.5'`. `int()` only takes whole numbers. `float("2.5")` would

work, but a customer cannot buy half a loaf.

When you cannot read the message

Some messages are not as plain as this one. This is one of the places the AI policy lets you ask AI: **paste the error message, never your program.** Ask what the message means. Then you go and find the line yourself, and you fix it yourself. Log it as you ask:

```
### 2026-10-05

- **Kind:** Error
- **My question:** What does `ValueError: invalid␣
↪literal for int() with base 10: 'two'` mean?
- **What I learned:** int() can only turn text into a␣
↪number if the text is digits, like '2'.
- **What I did next:** Found line 37 in my shop and␣
↪started this chapter's fix.
```

What an exception is

When a line cannot be done, Python *raises an exception*. Think of it as Python throwing its hands up: *I can't do this line.* If nothing in the program is ready to catch it, the program stops and prints the traceback.

> **For your glossary:** *exception* — a signal that a line could not be done. *Raise* — to send that signal. *Catch* (or *handle*) — to be ready for it and do something sensible instead of stopping.

Look at the banner's left side. The six structures are the six ways the flow can move. Command goes to the next line. Decision takes one branch or the other. Loop goes round again. The exception is the one that says: **something went wrong — leave this path and go handle it.**

try and except

Python lets you say: *try these lines, and if this kind of exception happens, do this instead.*

```
text = input("How many loaves? ")
try:
    count = int(text)
    print("You ordered", count)
except ValueError:
    print(text, "is not a whole number.")
```

Run it twice.

```
How many loaves? two
two is not a whole number.
```

```
How many loaves? 4
You ordered 4
```

Walk it with the flow in mind:

- `try:` means *the lines indented under me might fail; be ready*.
- If `count = int(text)` works, the flow carries on down: the `print` under it runs, and the `except` part is skipped, the way an `else` is skipped when the `if` was true.
- If `count = int(text)` raises a `ValueError`, the flow **leaves that path right there**. The `print("You ordered", count)` line never runs. The flow jumps to `except ValueError:` and runs the lines under it.
- Either way, the program does not stop. After the `try`/`except`, it carries on with whatever comes next.

  For your glossary: *try block*, *except block*.

⧖ **6.3)** In the run where the customer typed `two`, which lines ran?

Answer: Line 1 (`text = input(...)`), line 2 (`try:`), line 3 (`count = int(text)`, which failed), line 5 (`except ValueError:`) and line 6 (the `print`). Line 4 never ran. Notice also that `count` never got a value, because the assignment on line 3 never finished. If a later line used `count`, that would be a new error.

Name the exception you expect

You can write `except:` with nothing after it. Don't. It catches every exception, including the ones you did not expect, and then your friendly message lies. Suppose you misspell a variable inside the `try`:

```
try:
    count = int(txt)       # txt should be text
except:
    print("Please type a whole number.")
```

The customer types `3`, and the program says *Please type a whole number.* The real problem was a `NameError`, your typo, and the bare `except` hid it from you. **Catch the exception you expect. Let Python show you the ones you don't**, so you can fix them.

Keep asking

One message is better than a crash, but the customer still has not bought any bread. What we want is: *keep asking until a whole number is typed.* "Keep doing something while it's not right yet" is a loop, and you already have one: `while`.

```
count = 0
while count < 1:
    text = input("How many loaves? ")
    try:
        count = int(text)
        if count < 1:
            print("Please order at least 1.")
    except ValueError:
        print("Please type a whole number, like 3.")
print("You ordered", count)
```

```
How many loaves? two
Please type a whole number, like 3.
How many loaves? -2
Please order at least 1.
```

(continues on next page)

(continued from previous page)

```
How many loaves? 3
You ordered 3
```

Look at how the pieces fit:

- `count = 0` puts a value in the cup before the loop, so the test `count < 1` has something to look at. 0 is less than 1, so the loop runs at least once.
- If the customer types `two`, the `int()` fails, `count` keeps its 0, the message prints, and the test is still true: round again.
- If the customer types `-2`, `int()` works. `-2` is a perfectly good whole number. So `count` becomes -2, the `if` catches it, and since -2 is still less than 1, round again.
- If the customer types `3`, `count` becomes 3, the `if` is false, and the test `3 < 1` is false. The loop ends.

⧖ **6.4)** Why is `-2` caught by an `if` and not by the `except`?

Answer: Because nothing went wrong in Python. `int("-2")` is easy: it gives -2. The *rule* that you cannot buy fewer than one loaf is Tina's rule, not Python's. **Python's problems are exceptions. Your rules are decisions.** If the line can be done but the answer is not allowed, use an `if`.

This is a pattern you will use for the rest of your life as a programmer. Learn it well:

```
value = starting value that fails the test
while value fails the test:
    ask again
    try:
        convert it
        if it breaks your rule:
            say so
    except the error you expect:
        say so
```

Playing the memory game with an exception

Now be the computer. Here is the keep-asking program with line numbers:

```
 1  count = 0
 2  while count < 1:
 3      text = input("How many loaves? ")
 4      try:
 5          count = int(text)
 6          if count < 1:
 7              print("Please order at least 1.")
 8      except ValueError:
 9          print("Please type a whole number, like␣
↪3.")
10  print("You ordered", count)
```

The customer types `two`, then `3`. On your spreadsheet, each row is one sheet. Here is the run (Stack written bottom first; the top is the line you are on):

Shee	Line	Con-trol	Key-board	Screen shows (new)	coun	text	Stack
1	1	Com-mand			0		1
2	2	Loop			0		2
3	3	Com-mand	`two`	How many loaves? two	0	two	2, 3
4	4	Ex-cep-tion			0	two	2, 4
5	5	**Ex-cep-tion**			0	two	2, 5
6	8	Ex-cep-tion			0	two	2, 8
7	9	Com-mand		Please type a whole number, like 3.	0	two	2, 9
8	2	Loop			0	two	2
9	3	Com-mand	`3`	How many loaves? 3	0	3	2, 3
10	4	Ex-cep-tion			0	3	2, 4
11	5	Com-mand			**3**	3	2, 5
12	6	Deci-sion			3	3	2, 6
13	2	Loop			3	3	2
14	10	Com-mand		You ordered 3	3	3	10

Three things to notice.

The same line, two different controls. On sheet 5, line 5 raised an exception, so its Control is *Exception*: the flow left the path. On sheet 11, the very same line worked, so it is *Command*: do it, go to the next line. The sheet records what *happened*, not what the line looks like. That is the whole point

of playing the game.

Line 6 and line 7 are missing from the first time round. After the exception on line 5, the flow jumped straight to line 8. Lines 6 and 7 were skipped. If you had drawn sheets for them, your trace would be wrong, and you would have learned something false.

The loop's 2 sat on the Stack the whole time. The exception did not disturb it. The flow left the `try` path, but it was still inside the loop, and the loop still owed a return to line 2. Within one block of code, an exception behaves like a decision: it takes one path instead of another. (When you meet functions in chapter 9, an exception can do something stronger, but that is for later.)

⧖ **6.5)** How many sheets would you need if the customer typed `two`, then `-2`, then `3`?

Answer: **20.** Count it the way you would on paper, one round at a time:

Part of the run	Lines that run	Sheets
Before the loop	1	1
`two`	2, 3, 4, 5, 8, 9	6
`-2`	2, 3, 4, 5, 6, 7	6
`3`	2, 3, 4, 5, 6	5
The last test, and the end	2, 10	2

The `-2` round is the longest of the three, because `int()` works and then the `if` is true, so line 7 runs too. If you guessed instead of counting, you probably got it wrong. Most people do. That is why we write the sheets out.

Fixing the shop

Now put the pattern into Tina's shop. In your `shop.py` from chapter 5, find the line that asks for the count:

```
count = int(input(f"How many {item}? "))
```

and replace it with the keep-asking loop:

```
        count = 0
        while count < 1:
            text = input(f"How many {item}? ")
            try:
                count = int(text)
                if count < 1:
                    print("Please order at least 1.")
            except ValueError:
                print("Please type a whole number,␣
↪like 3.")
```

Watch the indentation. This whole block sits inside the shop's `while` loop and inside the `else` that runs when the choice is on the menu. That is a loop inside a loop. The inner loop keeps asking for a count; the outer loop keeps the shop open.

Here is the whole program as it stands now. Save it in your site as `shop_ch06.py`.

```
# shop.py - version 6: keep asking until the count is␣
↪a whole number
shop = "Tina's Corner Shop"
bread_price = 2.50
milk_price = 1.75
egg_price = 3.00
bread_stock = 12
milk_stock = 20
egg_stock = 10
day_total = 0.0
sales = 0

print(shop)
choice = ""
while choice != "Q":
    choice = input("B)read M)ilk E)ggs Q)uit: ").
↪upper()

    # look up the item
```

(continues on next page)

(continued from previous page)

```
    item = ""
    if choice == "B":
        item = "loaves of bread"
        price = bread_price
        stock = bread_stock
    elif choice == "M":
        item = "bottles of milk"
        price = milk_price
        stock = milk_stock
    elif choice == "E":
        item = "cartons of eggs"
        price = egg_price
        stock = egg_stock

    if choice == "Q":
        print("Closing time.")
    elif item == "":
        print("Sorry, we don't sell that.")
    else:
        count = 0
        while count < 1:
            text = input(f"How many {item}? ")
            try:
                count = int(text)
                if count < 1:
                    print("Please order at least 1.")
            except ValueError:
                print("Please type a whole number,␣
↪like 3.")
        if count > stock:
            print(f"Sorry, only {stock} left.")
        else:
            total = count * price
            if count >= 10:
                total = total * 0.9
                print("10% off for 10 or more!")
            stock = stock - count
```

(continues on next page)

(continued from previous page)

```
            if choice == "B":
                bread_stock = stock
            elif choice == "M":
                milk_stock = stock
            else:
                egg_stock = stock
            day_total = day_total + total
            sales = sales + 1
            print(f"Total: {total:.2f}")

# the day's summary
print()
print(f"Sales today: {sales}")
print(f"Taken in: {day_total:.2f}")
print(f"Left: bread {bread_stock}, milk {milk_stock},␣
↪eggs {egg_stock}")
```

A run with a difficult customer:

```
Tina's Corner Shop
B)read M)ilk E)ggs Q)uit: b
How many loaves of bread? two
Please type a whole number, like 3.
How many loaves of bread? 3
Total: 7.50
B)read M)ilk E)ggs Q)uit: m
How many bottles of milk? 0
Please order at least 1.
How many bottles of milk? 12
10% off for 10 or more!
Total: 18.90
B)read M)ilk E)ggs Q)uit: e
How many cartons of eggs? 11
Sorry, only 10 left.
B)read M)ilk E)ggs Q)uit: x
Sorry, we don't sell that.
B)read M)ilk E)ggs Q)uit: q
```

(continues on next page)

(continued from previous page)

```
Closing time.

Sales today: 2
Taken in: 26.40
Left: bread 9, milk 8, eggs 10
```

The shop stays open no matter what the customer types. Look at the last three menu choices: a wrong letter (`x`) and an order that is too big (`11` eggs) were both caught by **decisions**, because Python had no trouble with them. Only `two` needed an **exception**.

⧖ **6.6)** What is the only difference between `shop_ch05.py` and `shop_ch06.py`?

Answer: The one line that asked for the count. It became the keep-asking loop, and the prompt moved inside it. Everything else is chapter 5's program, line for line. That is how a program grows: one change at a time, each one tested.

Design: what could go wrong?

You have been using stepwise refinement: code a little, test, refine. Add one question to every test from now on: **what could a person type here that would break this?** For every `input()`, try:

- a word instead of a number;
- a number with a decimal point;
- a negative number, and zero;
- nothing at all (just press Enter).

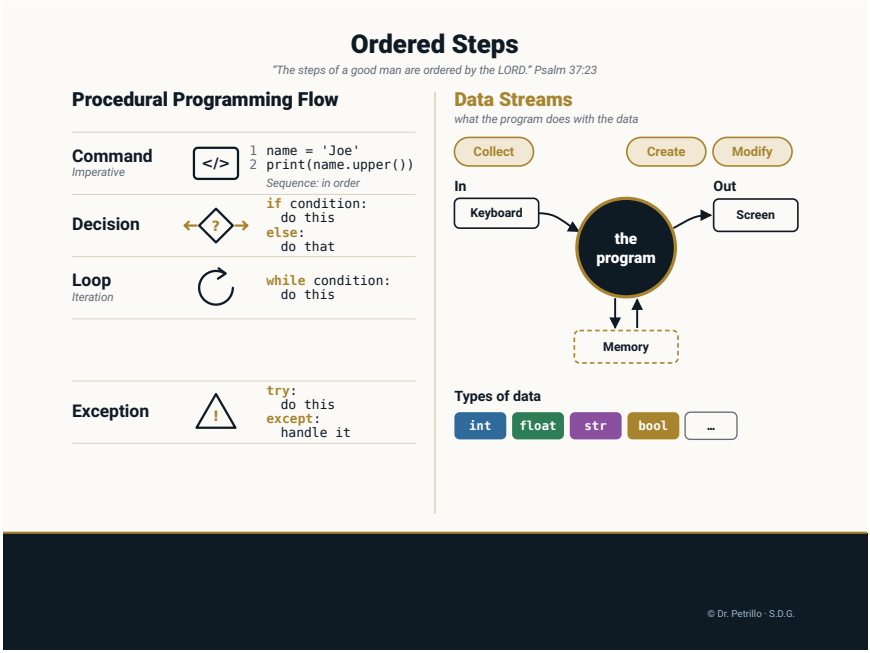
Each one is either something Python cannot do (catch the exception) or something your rules forbid (make a decision).

Grace at the counter

A good shopkeeper does not shout at the customer who says “two” instead of holding up two fingers. She smiles and asks again. Our program now does the same thing. I find that worth noticing. We all get things wrong, many times a day, and the world would be a hard place if every mistake shut the shop. We are shown patience far more often than we deserve. Building a little patience into a program is a small way of passing it on.

Your picture so far

You have added the fifth structure. Five of the six ways a program moves its data are now yours.



Exercises

Do these in order. Each one is a little harder than the one before. Write every program yourself, and play the memory game on at least exercises 4 and 8.

1. **Name the error.** For each line, say whether it is a syntax, runtime or

logic error.

- (a) `print("Bread: " + 2.50)`
- (b) `average = total / count` when `count` is 0
- (c) `if count > 10` (no colon)
- (d) `total = price * count + 1` when the customer should pay `price * count`

2. **Read the traceback.** What kind of error is this, which line failed, and why?

```
Traceback (most recent call last):
  File "prices.py", line 4, in <module>
    price = float(text)
ValueError: could not convert string to float:
↪'2,50'
```

3. **Predict, then run.** What does each of these print, or what error does it raise? `int("7")`, `int(" 7 ")`, `int("3.5")`, `float("3.5")`, `int("l")`.

4. **One catch.** Write a program that asks for a person's age, converts it with `int()`, and prints `Next year you will be` and the age plus one. If they type something that is not a whole number, print `That is not a whole number.` instead of crashing.

5. **Keep asking.** Change exercise 4 so it keeps asking until a whole number is typed.

6. **Your rule and Python's.** Change exercise 5 so an age below 0 or above 120 is refused with `Please type a real age.` Which of the two problems is an exception, and which is a decision?

7. **A price.** Write a program that keeps asking for a price until a number is typed (use `float()`) that is not negative. Print it with two decimal places.

8. **The memory game.** Trace your program from exercise 7 on the spreadsheet with the inputs `abc`, `-1`, `2.5`. How many sheets did it take? Which sheets have *Exception* as their Control?

9. **Not every check needs try.** Write a program that keeps asking `Another sale? (Y/N)` until the person types `Y` or `N` (upper or lower case). Why does this one need no `try` at all?

10. **Three tries.** Change the shop's keep-asking loop so a customer gets at most three tries. After three bad answers, print `Sorry, let's start again.` and go back to the menu without making a sale. (Hint: count the tries, and make the `while` test check two things.)

Practicing with AI

When you want more practice, ask AI to make more programs for you to write. Start like this:

> *I need help practicing try and except in Python, and loops that keep asking until the input is right. Starting simple, give me programs that I can write. Do not use lists. If I get stuck I'll ask for a hint. Then increase their complexity so I get better at designing programs that do not crash when people type the wrong thing.*

Write every program yourself, and log the conversation as you go.

Words for your glossary

Write each of these in your own words in your site's glossary: **syntax error**, **runtime error**, **logic error**, **traceback**, **exception**, **raise**, **catch**, **handle**, **try block**, **except block**.

Summary

- Three kinds of errors: **syntax** (Python cannot read it; nothing runs), **runtime** (Python cannot do a line while running), **logic** (it runs and gives the wrong answer).
- Read a **traceback** from the bottom up: the kind of error and the message, then the line, then where.
- You may paste an **error message** to AI. Never your program.
- An **exception** is the fifth structure: something went wrong, so leave this path and go handle it.

- `try:` the lines that might fail; `except ValueError:` what to do if they do. Name the exception you expect.
- **Python's problems are exceptions. Your rules are decisions.**
- Keep asking: start with a value that fails the test, then `while` + `try` + `if` until the answer is right.
- On the sheet, the same line can be *Command* one time and *Exception* the next. Record what happened.

Answers

1. (a) Runtime: `+` cannot join a `str` and a `float` (a `TypeError`). (b) Runtime: a `ZeroDivisionError`. (c) Syntax: the `if` line must end with a colon. (d) Logic: it runs, but the customer is charged one too much.
2. A `ValueError` on line 4 of `prices.py`. `float()` was given `'2,50'`, and Python uses a point, not a comma, for decimals.
3. `7`; `7` (spaces around the digits are allowed); `ValueError` (`int()` will not take a decimal point); `3.5`; `ValueError` (a lowercase L is not a one).
4.
```
text = input("How old are you? ")
try:
    age = int(text)
    print("Next year you will be", age + 1)
except ValueError:
    print("That is not a whole number.")
```
5.
```
age = -1
while age < 0:
    text = input("How old are you? ")
    try:
        age = int(text)
    except ValueError:
        print("That is not a whole number.")
print("Next year you will be", age + 1)
```

Starting `age` at -1 makes the test true, so the loop runs at least once.

6.
```
age = -1
while age < 0 or age > 120:
    text = input("How old are you? ")
    try:
        age = int(text)
        if age < 0 or age > 120:
            print("Please type a real age.")
    except ValueError:
        print("That is not a whole number.")
print("Next year you will be", age + 1)
```

A word is an exception: `int()` cannot do it. An age of 200 is a decision: `int("200")` works fine; it is your rule that refuses it.

7.
```
price = -1.0
while price < 0:
    text = input("Price: ")
    try:
        price = float(text)
        if price < 0:
            print("A price cannot be negative.")
    except ValueError:
        print("Please type a number, like 2.50.")
print(f"The price is {price:.2f}")
```

With `abc`, `-1`, `2.5` it prints the two messages and then `The price is 2.50`.

8. With the program in answer 7 laid out as 10 lines (line 10 being the last `print`): 1; then `abc`: 2, 3, 4, 5, 8, 9; then `-1`: 2, 3, 4, 5, 6, 7; then `2.5`: 2, 3, 4, 5, 6; then 2, 10. That is **20 sheets**. Control is *Exception* on the `try` lines, on line 5 the first time round (where `float("abc")` failed), and on line 8.

9.
```
answer = ""
while answer != "Y" and answer != "N":
    answer = input("Another sale? (Y/N) ").upper()
print("You said", answer)
```

Nothing here can make Python fail: `input()` always gives a string, and `.upper()` works on any string. Refusing `maybe` is your rule, so a decision (the `while` test) is enough.

10. Inside the shop, replace the keep-asking block with:

```
        count = 0
        tries = 0
        while count < 1 and tries < 3:
            text = input(f"How many {item}? ")
            tries = tries + 1
            try:
                count = int(text)
                if count < 1:
                    print("Please order at least 1.")
            except ValueError:
                print("Please type a whole number, like 3.")
        if count < 1:
            print("Sorry, let's start again.")
        else:
            (the rest of the sale, indented one more step)
```

The `while` stops either when a good count is typed or after the third try. The `if` after it tells which one happened.

A line for every loaf

Tina wants better receipts. When a customer buys three loaves, she wants the receipt to show three lines, one for each loaf, the way a shop till does:

```
--------------------
  Bread  2.50
  Bread  2.50
  Bread  2.50
--------------------
  Total  7.50
```

You can already do this. You know `while`. Count from 0, and keep printing while the counter is less than the number of loaves:

```
count = 3
i = 0
while i < count:
    print("  Bread  2.50")
    i = i + 1
```

⧖ **7.1)** What are the three things this loop needs to work?

Answer: A counter that starts somewhere (`i = 0`), a test that stops it (`i < count`), and a line that moves the counter on (`i = i + 1`). Forget the last one and you have an infinite loop. You met that in chapter 5.

Counting like this is so common that Python gives you a loop that does the counting for you. It is called `for`.

```
count = 3
for i in range(count):
    print("  Bread  2.50")
```

```
  Bread  2.50
  Bread  2.50
  Bread  2.50
```

Same output. Same flow: round and round, then on. It is still the loop on the banner, not a new structure. The difference is who looks after the counter. With `while`, you do. With `for`, Python does.

> **For your glossary:** *for loop* — a loop that runs its body once for each item it is given.

`while` or `for`?

Here is the rule I give my students:

- **`while`**: *keep going until something changes.* You don't know how many times. The shop stays open until someone presses `Q`. You keep asking until a whole number is typed.
- **`for`**: *once for each one.* You know what you are going through: three loaves, five numbers, every letter of a name.

Both are loops. Pick the one that says what you mean.

⧖ **7.2)** Which loop would you use for each of these? (a) Ask for a password until it is right. (b) Print a line for each of the 12 cartons of eggs. (c) Keep a game going until the player has no lives left.

Answer: (a) `while`: you do not know how many tries it will take. (b) `for`: twelve times, once for each. (c) `while`: you keep going until something changes.

Counting with `range`

`range` makes the numbers for a `for` loop to walk through.

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

⧖ **7.3)** Why does it stop at 4 and not 5?

Answer: `range(5)` means *five numbers, starting at 0*. Computers count from 0. You will see why in a moment, when we look at the letters in a word. For now: `range(5)` gives 0, 1, 2, 3, 4. Five numbers. The 5 itself is the stopping point, and it is never included.

`range` can take up to three numbers:

You write	You get	Read it as
`range(5)`	0 1 2 3 4	five numbers from 0
`range(1, 6)`	1 2 3 4 5	from 1, stop before 6
`range(0, 10, 2)`	0 2 4 6 8	from 0, stop before 10, in steps of 2
`range(5, 0, -1)`	5 4 3 2 1	from 5, stop before 0, counting down
`range(3, 3)`	nothing	from 3, stop before 3: no numbers at all

> **For your glossary:** *range* — the numbers from a start up to, but not including, a stop, in steps.

To print them on one line, tell `print` not to end the line. Normally `print` puts a new line at the end. `end=" "` puts a space there instead:

```
for i in range(1, 6):
    print(i, end=" ")
print()
```

```
1 2 3 4 5
```

The empty `print()` at the end finishes the line.

A word is letters in a row

Now the second example of this book: **words.**

A string is not one thing in a cup. It is letters in a row, and Python lets you walk through them, one for each:

```
name = "Tina"
for letter in name:
    print(letter)
```

```
T
i
n
a
```

Read the first line of the loop the way it sounds: *for each letter in name.* Each time round, the next letter is put in the cup called `letter`, and the body runs once.

Each letter has a position, called its *index*. The first letter is at index 0:

Index	0	1	2	3
Letter	`T`	`i`	`n`	`a`

> **For your glossary:** *index* — the position of a letter in a string, counting from 0.

```
name = "Tina"
print(len(name))
print(name[0])
print(name[3])
```

```
4
T
a
```

`len()` gives the *length*: how many letters. `name[0]` is the letter at index 0. Now you can see why `range` starts at 0: `range(len(name))` gives exactly

the indexes of the letters, 0 to 3.

```
for i in range(len(name)):
    print(i, name[i])
```

```
0 T
1 i
2 n
3 a
```

⧖ **7.4)** What happens if you ask for `name[4]`?

Answer: `IndexError: string index out of range`. "Tina" has four letters, at indexes 0 to 3. There is no index 4. You met exceptions in the last chapter. Here is a new one, and you know how to read its traceback.

Playing the memory game with `for`

Be the computer for this program:

```
1  name = "Tina"
2  for letter in name:
3      print(letter)
```

Sheet	Line	Control	Screen shows (new)	`name`	`letter`	Stack
1	1	Command		Tina		1
2	2	Loop		Tina	T	2
3	3	Command	T	Tina	T	2, 3
4	2	Loop		Tina	i	2
5	3	Command	i	Tina	i	2, 3
6	2	Loop		Tina	n	2
7	3	Command	n	Tina	n	2, 3
8	2	Loop		Tina	a	2
9	3	Command	a	Tina	a	2, 3
10	2	Loop		Tina	a	2

Look at the `letter` column. **One cup, rewritten four times.** It does not hold the whole name; it holds one letter at a time, and each new letter replaces the old one, just as `x = x + 7` replaced the 1 in the egg cup back in chapter 2.

Look at sheet 10. The flow came back to line 2 to get the next letter, found there were none left, and the loop ended. `letter` still holds `a`, the last thing put in it. The Stack's 2 waited under line 3 every time, because the loop owed a return to its top line. When the letters ran out, the debt was paid and the loop was done.

⧖ **7.5)** How many sheets would `for letter in "Bread":` with a `print` in the body take, counting the line before it?

Answer: 1 for the line before, 2 sheets for each of 5 letters (10), and 1 last sheet when line 2 finds no more letters: **12**.

The memory game moves to the computer

Look at what you just did: ten sheets for a three-line program. Words have many letters, and your shop now prints a line for every loaf. The sheets are multiplying, as they did when you met `while`. That is when we change tools.

From now on, you play the memory game in **`memory_game.html`**. Download it here: `memory_game.html`. It is one file; save it in your site and open it in your web browser. It needs no internet and installs nothing.

It looks like the sheet you have been filling in, because it *is* that sheet:

- **Line #, Code, Comment, Control, Keyboard.** You type the answers.
- **Screen** and **File.** Leave File empty until the next chapter.
- **Stack.** Push a line number when the flow owes a return to it; pop it when it doesn't.
- **Memory.** A grid of cells named A1 to H3, the same names as your spreadsheet. A cell that changed since the last sheet turns amber, so you see at a glance what a line did.
- **Data dictionary.** Name, type, location (which cell) and scope. For now the scope is the line the variable was first given a value, then `G`, like `1:G`.

Press **Save sheet**, and the sheet is drawn above, and the next one starts from it. **Download** saves the page with all your sheets in it. That file is your work: keep it in your site, and hand it in when you are asked.

One thing it will never do: **it never reads your program.** It does not know what line 4 does. You tell it. The tool takes the drawing off your hands, not the thinking. Working out what the line did is still the assignment.

Look at the **Control** list. It has all six structures. You know four of them now: Command, Decision, Loop, Exception. The other two, Function and Recursion, are coming. When you get there, you will see you have already been practising for them.

Words: four worked examples

Initials

Tina writes the customer's initials on special orders. Given a full name, take the first letter, and every letter that comes right after a space.

```
name = input("Your full name: ")
initials = name[0]
after_space = False
for letter in name:
    if after_space:
        initials = initials + letter
    after_space = letter == " "
print("Initials:", initials)
```

```
Your full name: Tina Okafor
Initials: TO
```

```
Your full name: Ana Maria Lopez
Initials: AML
```

The trick is the cup called `after_space`. It holds `True` or `False`: *was the letter before this one a space?* Each time round, we use it first, then set it for the next letter. That line, `after_space = letter == " "`, puts a `bool` in the cup: `True` if this letter is a space, `False` if it is not.

⧖ **7.6)** Why does `initials` start with `name[0]` instead of an empty string?

Answer: Because the first letter has no space before it. Nothing in the loop would ever catch it.

Counting vowels

```
name = input("Your name: ")
vowels = 0
for letter in name.lower():
    if letter == "a" or letter == "e" or letter == "i"␣
↪or letter == "o" or letter == "u":
```

(continues on next page)

(continued from previous page)

```
        vowels = vowels + 1
print(name, "has", vowels, "vowels.")
```

```
Your name: Tina Okafor
Tina Okafor has 5 vowels.
```

`name.lower()` makes a lowercase copy to walk through, so `O` counts as well as `o`. The original `name` is not changed.

That long `if` works, but Python has a shorter way to ask *is this letter one of these?*:

```
print("a" in "aeiou")
print("t" in "aeiou")
```

```
True
False
```

So the test can be `if letter in "aeiou":`. Same decision, shorter to write.

The first word

Here is one from my classroom. Someone types a sentence. Find the first word.

The letters go into a cup one at a time, as long as they are not a space. When you hit a space, stop. “As long as” and “stop” — which loop is that?

```
sentence = input("Say something: ")
first = ""
i = 0
while i < len(sentence) and sentence[i] != " ":
    first = first + sentence[i]
    i = i + 1
print("The first word is", first)
```

```
Say something: Fresh bread today
The first word is Fresh
```

It is `while`, because we stop when something changes, which is when we reach a space. Notice the test checks two things: we have not run out of letters, *and* this letter is not a space. If someone types one word with no spaces at all, the first check stops the loop at the end of the word.

You can do it with `for`, but `for` always walks every letter. You need a cup that remembers you have already passed the space:

```
sentence = input("Say something: ")
first = ""
found_space = False
for letter in sentence:
    if letter == " ":
        found_space = True
    if not found_space:
        first = first + letter
print("The first word is", first)
```

Both give `Fresh`. ⧖ **7.7)** Which one does less work on `Fresh bread today`?

Answer: The `while` version. It stops at the first space, after five letters. The `for` version looks at all seventeen characters, even though it stopped adding after five. For a short sentence it doesn't matter. The point is that the loop you choose says what you mean: *until* or *each*.

Checking a coupon, letter by letter

Tina gives out a coupon code, `SAVE10`. The customer types it at the till.

```
coupon = "SAVE10"
code = input("Coupon code: ").upper()
match = len(code) == len(coupon)
if match:
    for i in range(len(coupon)):
        if code[i] != coupon[i]:
            match = False
if match:
    print("10% off!")
else:
```

(continues on next page)

(continued from previous page)

```
    print("Sorry, that code does not work.")
```

```
Coupon code: save10
10% off!
```

```
Coupon code: SAVE1O
Sorry, that code does not work.
```

The second customer typed the letter O instead of a zero. Every letter matches until index 5, where `"O"` is not `"0"`, and `match` becomes `False`.

You may be thinking: couldn't I just write `if code == coupon:`? Yes, and in your own programs you should. But what do you think == does inside, when it compares two strings? Exactly this: same length, then letter by letter. Now you know what is behind it. That is the difference between pushing buttons and understanding.

Tina's receipt

Back to the shop. When a sale goes through, the receipt shows a line of dashes, one line for each item, another line of dashes, and the total. First, the receipt on its own:

```
item = "Bread"
price = 2.50
count = 3
for i in range(20):
    print("-", end="")
print()
for i in range(count):
    print(f"  {item}  {price:.2f}")
for i in range(20):
    print("-", end="")
print()
print(f"  Total  {count * price:.2f}")
```

```
--------------------
  Bread  2.50
  Bread  2.50
  Bread  2.50
--------------------
  Total  7.50
```

(Python has a shortcut for the dashes: `print("-" * 20)`. Use it once you have written the loop yourself and know what it does.)

To put this into the shop, the receipt needs a short name for each line. The `item` cup holds "loaves of bread", which reads well in a question but not on a till roll. Where the shop looks up the item, add a `label` cup:

```
    if choice == "B":
        item = "loaves of bread"
        label = "Bread"
        price = bread_price
        stock = bread_stock
    elif choice == "M":
        item = "bottles of milk"
        label = "Milk"
        price = milk_price
        stock = milk_stock
    elif choice == "E":
        item = "cartons of eggs"
        label = "Eggs"
        price = egg_price
        stock = egg_stock
```

and replace `print(f"Total: {total:.2f}")` with the receipt:

```
            for i in range(20):
                print("-", end="")
            print()
            for i in range(count):
                print(f"  {label}  {price:.2f}")
            for i in range(20):
                print("-", end="")
```

(continues on next page)

(continued from previous page)

```
            print()
            print(f"  Total  {total:.2f}")
```

Here is the whole program. Save it as `shop_ch07.py`.

```
# shop.py - version 7: a receipt, one line for each␣
↪item
shop = "Tina's Corner Shop"
bread_price = 2.50
milk_price = 1.75
egg_price = 3.00
bread_stock = 12
milk_stock = 20
egg_stock = 10
day_total = 0.0
sales = 0

print(shop)
choice = ""
while choice != "Q":
    choice = input("B)read M)ilk E)ggs Q)uit: ").
↪upper()

    # look up the item
    item = ""
    if choice == "B":
        item = "loaves of bread"
        label = "Bread"
        price = bread_price
        stock = bread_stock
    elif choice == "M":
        item = "bottles of milk"
        label = "Milk"
        price = milk_price
        stock = milk_stock
    elif choice == "E":
        item = "cartons of eggs"
```

(continues on next page)

(continued from previous page)

```
        label = "Eggs"
        price = egg_price
        stock = egg_stock

    if choice == "Q":
        print("Closing time.")
    elif item == "":
        print("Sorry, we don't sell that.")
    else:
        count = 0
        while count < 1:
            text = input(f"How many {item}? ")
            try:
                count = int(text)
                if count < 1:
                    print("Please order at least 1.")
            except ValueError:
                print("Please type a whole number,␣
↪like 3.")
        if count > stock:
            print(f"Sorry, only {stock} left.")
        else:
            total = count * price
            if count >= 10:
                total = total * 0.9
                print("10% off for 10 or more!")
            stock = stock - count
            if choice == "B":
                bread_stock = stock
            elif choice == "M":
                milk_stock = stock
            else:
                egg_stock = stock
            day_total = day_total + total
            sales = sales + 1
            for i in range(20):
                print("-", end="")
```

(continues on next page)

(continued from previous page)

```
            print()
            for i in range(count):
                print(f"  {label}  {price:.2f}")
            for i in range(20):
                print("-", end="")
            print()
            print(f"  Total  {total:.2f}")

# the day's summary
print()
print(f"Sales today: {sales}")
print(f"Taken in: {day_total:.2f}")
print(f"Left: bread {bread_stock}, milk {milk_stock},␣
↪eggs {egg_stock}")
```

A run:

```
Tina's Corner Shop
B)read M)ilk E)ggs Q)uit: b
How many loaves of bread? 3
--------------------
  Bread  2.50
  Bread  2.50
  Bread  2.50
--------------------
  Total  7.50
B)read M)ilk E)ggs Q)uit: e
How many cartons of eggs? 2
--------------------
  Eggs  3.00
  Eggs  3.00
--------------------
  Total  6.00
B)read M)ilk E)ggs Q)uit: q
Closing time.

Sales today: 2
```

(continues on next page)

(continued from previous page)

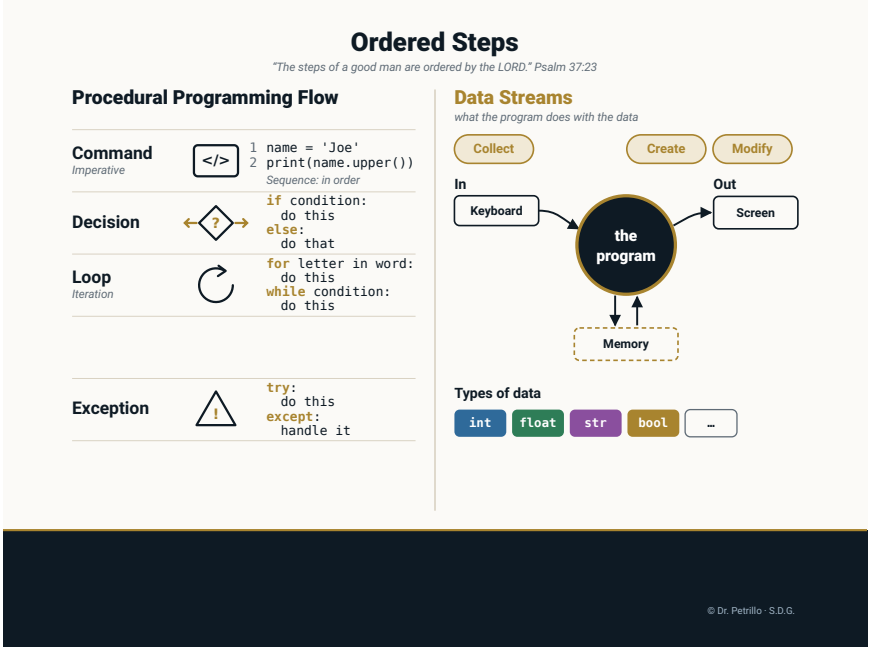
```
Taken in: 13.50
Left: bread 9, milk 20, eggs 8
```

⧖ **7.8)** A customer buys 12 milks and gets the 10% discount. What is wrong with the receipt?

Answer: Above the receipt the screen says `10% off for 10 or more!`, but the receipt itself shows twelve lines of `Milk 1.75`, which add up to 21.00, and then `Total 18.90`. The receipt does not add up. That is a logic error of the polite kind: nothing crashes, but the customer is confused. Exercise 10 fixes it.

Your picture so far

The Loop now has its second form. You have `while` for *until* and `for` for *each*.



Exercises

Write every program yourself. Play the memory game in `memory_game.html` on at least exercises 3 and 7, and download your sheets into your site.

1. **Predict.** Write down what each prints, then run it to check: `range(4)`, `range(2, 7)`, `range(10, 0, -2)`, `range(1, 10, 3)`. (Use a `for` loop with `print(i, end=" ")`.)
2. **Stars.** Ask for a number and print that many `*` on one line, using a `for` loop.
3. **Spell it.** Ask for a word and print each letter on its own line, with its index in front: `0 b`, `1 r`, and so on.
4. **Sum to n.** Ask for a whole number `n` (keep asking until it is one, as in chapter 6) and add up 1 + 2 + … + n with a `for` loop. For 10 it should print 55.
5. **Count the spaces.** Ask for a sentence and say how many words it has, by counting spaces. (Assume single spaces between words.)
6. **Every other letter.** Ask for a word and print every second letter, starting with the first, using `range` with a step of 2.
7. **Backwards.** Ask for a word and print it backwards. Do it two ways: (a) build a new string, putting each letter in *front* of what you have so far; (b) use `range` to count the indexes down from the last one to 0.
8. **Shout the name.** Ask for a name and print it with a space between each letter, in capitals: `T I N A`.
9. **Coupons.** Change the coupon checker so the code is also refused if it has the right length but starts with a space. Then add a second code, `BREAD5`. Which is clearer: two letter-by-letter checks, or `==`? Now that you know what `==` does, use it.
10. **An honest receipt.** In `shop_ch07.py`, when the discount applies, print a line `10% off` and the amount taken off before the total, so the receipt adds up.

Practicing with AI

> *I need help practicing for loops in Python: range() and walking through the letters of a string. Starting simple, give me programs that I can write. Do not use lists. If I get stuck I'll ask for a hint. Then increase their complexity so I get better at designing programs with loops.*

Write every program yourself. Log the conversation as you go.

Words for your glossary

Write each of these in your own words in your site's glossary: **for loop**, **range**, **index**.

Summary

- **`while`** is for *until something changes*. **`for`** is for *once for each one*. Both are the Loop on the banner.
- `for i in range(n):` runs the body `n` times, with `i` from 0 to `n - 1`.
- `range(start, stop, step)`: from `start`, stopping **before** `stop`, in steps of `step` (negative counts down).
- `for letter in word:` puts each letter, in turn, into the same cup.
- `len(word)` is the number of letters. `word[i]` is the letter at index `i`; the first is `word[0]`, the last is `word[len(word) - 1]`. Past the end: `IndexError`.
- `letter in "aeiou"` asks whether the letter is one of those.
- `print(..., end="")` keeps the next `print` on the same line.
- **The memory game is now `memory_game.html`** (`download it`). It draws the sheet. It never reads your program; you do.

Answers

1. `0 1 2 3`; `2 3 4 5 6`; `10 8 6 4 2`; `1 4 7`.

2. ```
   n = int(input("How many stars? "))
   for i in range(n):
       print("*", end="")
   print()
   ```

3. ```
   word = input("A word: ")
   for i in range(len(word)):
       print(i, word[i])
   ```

4. ```
   n = 0
   while n < 1:
       try:
           n = int(input("A whole number: "))
       except ValueError:
           print("Please type a whole number.")
   total = 0
   for i in range(1, n + 1):
       total = total + i
   print(total)
   ```

   `range(1, n + 1)`: stop *before* `n + 1`, so `n` itself is included.

5. ```
   sentence = input("A sentence: ")
   spaces = 0
   for letter in sentence:
       if letter == " ":
           spaces = spaces + 1
   print(spaces + 1, "words")
   ```

   For `Fresh bread today` it prints `3 words`: two spaces, three words.

6. ```
   word = input("A word: ")
   for i in range(0, len(word), 2):
       print(word[i])
   ```

7. (a)
   ```

```
word = input("A word: ")
backwards = ""
for letter in word:
    backwards = letter + backwards
print(backwards)
```

For `bread` it prints `daerb`. Each new letter goes in front, so the last letter ends up first. (b)

```
word = input("A word: ")
for i in range(len(word) - 1, -1, -1):
    print(word[i], end="")
print()
```

Start at the last index, `len(word) - 1`; stop *before* -1, so 0 is included; step -1.

8.
```
name = input("Your name: ").upper()
for letter in name:
    print(letter, end=" ")
print()
```

9. The first check becomes `match = len(code) == len(coupon) and code[0] != " "`. (Think: what happens if `code` is empty? `code[0]` would raise an `IndexError`. Because `and` stops as soon as the first part is `False`, an empty code never reaches `code[0]`.) With two codes, `if code == "SAVE10" or code == "BREAD5":` is far clearer. Write the loop to understand it; use == to get work done.

10. Keep the total before the discount in its own cup:

```
total = count * price
discount = 0
if count >= 10:
    discount = total * 0.1
    total = total - discount
```

and before printing the total on the receipt:

```
            if discount > 0:
                print(f"  10% off  -{discount:.2f}
↪")
```

For 12 milks the receipt shows twelve lines of 1.75, then `10% off -2.10`, then `Total 18.90`.

The shop forgets

Tina sells bread all morning. At lunchtime the computer is switched off to save power. In the afternoon she starts the program again, sells more, presses `Q`, and the program says:

```
Closing time.

Sales today: 4
Taken in: 21.00
Left: bread 10, milk 16, eggs 7
```

That is the afternoon only. The morning is gone.

Why? Think back to the egg carton. Every variable is a cup in **memory**, and memory only lasts as long as the program runs. When the program ends, the carton is emptied. Switch off the power and there is not even a carton.

> **For your glossary:** *memory* (the computer's working memory) — where a running program keeps its variables. It is emptied when the program ends.

If we want something to last, we have to write it down somewhere that is not emptied. That somewhere is a **file**.

> **For your glossary:** *file* — data kept on a disk under a name, like `sales.txt`. It stays there after the program ends, after the power goes off, until someone changes or deletes it.

Look at the banner again. On the left of "the program" are the doors data comes **in** through. On the right are the doors it goes **out** through. You have used Keyboard in and Screen out since chapter 3. Both sides have one more door: **File**. This chapter opens them.

Writing: data goes out

Here is how the program writes one sale to a file:

```
total = 7.50
with open("sales.txt", "a") as f:
    f.write(f"{total:.2f}\n")
```

Run it, then open `sales.txt` in your editor. It holds one line:

```
7.50
```

Run it again, and look again:

```
7.50
7.50
```

Take it apart one piece at a time.

- **`open("sales.txt", "a")`** opens the file called `sales.txt`, in the same folder as your program. The `"a"` means *append*: add to the end of whatever is already there. If there is no such file yet, it is made.
- **`as f`** puts the opened file in a cup called `f`. What is in that cup is not the file's words. It is a *file object*: a connection to the file that remembers where you are in it.
- **`with ... :`** and the indented block under it mean *use the file for these lines, then close it*. When the block ends, Python closes the file for you, even if something goes wrong inside.
- **`f.write(...)`** writes a string into the file. Only a string: that is why the total is turned into text with the f-string.
- **`\n`** is the *newline*: the character that ends a line. `print` adds one for you. `write` does not. Leave it off and every sale lands on the same line: `7.507.507.50`.

  For your glossary: *open*, *close*, *file object*, *append*, *newline* (`\n`).

⧖ **8.1)** Why does it matter that the file is closed?

Answer: While a file is open, what you wrote may still be waiting in memory on its way to the disk. Closing it makes sure it is all written down, and lets other programs use the file. `with` closes it for you, so you can't forget. That is why we always use `with`.

Three ways to open

Mode	Means	If the file is there	If it is not
`"r"`	read	read from the start	`FileNotFoundError`
`"w"`	write	**empty it first**, then write	make it
`"a"`	append	write at the end	make it

`"r"` is what you get if you leave the mode out: `open("sales.txt")` means read.

⧖ **8.2)** Tina's program used `"w"` for the sales file instead of `"a"`. What would happen?

Answer: Every sale would empty the file first and then write itself. At the end of the day the file would hold one line: the last sale. Try it:

```
total = 6.00
with open("sales.txt", "w") as f:
    f.write(f"{total:.2f}\n")
```

```
6.00
```

The two 7.50 lines are gone. No warning, no undo. **`"w"` is for starting fresh. `"a"` is for adding.** Choose on purpose.

Reading: data comes in

At closing, Tina wants the day's total from the file: every sale, morning and afternoon. Suppose `sales.txt` holds:

```
7.50
6.00
18.90
```

```
day_total = 0
sales = 0
with open("sales.txt", "r") as f:
    for line in f:
        day_total = day_total + float(line.strip())
        sales = sales + 1
print(f"{sales} sales, {day_total:.2f} in all")
```

```
3 sales, 32.40 in all
```

`for line in f:` reads the file one line at a time, the way `for letter in word:` walked a word one letter at a time. Once for each line. Each time round, the next line is put in the cup called `line`.

Each line is a **string**, and it still has its newline on the end: `"7.50\n"`. `line.strip()` makes a copy with the spaces and newlines taken off both ends, giving `"7.50"`. Then `float()` turns it into the number 7.5, and we add it on.

> **For your glossary:** *strip* — remove spaces and newlines from both ends of a string.

⧖ **8.3)** Notice there is no `try` around `float()`. What would happen if someone had typed `seven` into the file by hand?

Answer: `ValueError: could not convert string to float: 'seven'`, and the program would stop. A file is input, the same as the keyboard. Your program can't trust a file any more than it trusts a customer. In this chapter our program is the only one writing `sales.txt`, so we accept the risk. Exercise 9 shows you what to do about it.

Watching the ▶ move

A file object remembers where you are in the file. In `memory_game.html` that is the **File** region: the file's name, its lines, and a ▶ on the line the next read will get.

Tina keeps her prices in a file, so she can change them without touching the program. `prices.txt` holds three lines, always in this order: bread, milk, eggs.

```
2.50
1.75
3.00
```

`f.readline()` reads **one** line: the line the ▶ is on. Then it moves the ▶ down one.

```
1  with open("prices.txt") as f:
2      bread_price = float(f.readline())
3      milk_price = float(f.readline())
4      egg_price = float(f.readline())
5  print(bread_price, milk_price, egg_price)
```

Play it in `memory_game.html`. Put `prices.txt` in the File region's name, and the three lines in its lines. Here are the sheets:

Shee	Line	Con-trol	File ▶ is on	bread_p	milk_p	egg_pr	Screen shows (new)
1	1	Com-mand	line 1 (`2.50`)				
2	2	Com-mand	line 2 (`1.75`)	2.5			
3	3	Com-mand	line 3 (`3.00`)	2.5	1.75		
4	4	Com-mand	**end of file**	2.5	1.75	3.0	
5	5	Com-mand	(closed)	2.5	1.75	3.0	2.5 1.75 3.0

In the html, type the next line number in the **next line ▶** box: 1 on the first sheet, then 2, then 3, then 4. There are only three lines, so 4 means *one past the last line*, which the sheet draws as ▶ **end of file**.

⧖ **8.4)** The Screen shows `2.5`, not `2.50`. Is something wrong?

Answer: No. `float("2.50")` is the number two and a half, and Python prints that as `2.5`. The `0` on the end was only how the text was written. When you want two decimal places for money, ask for them: `f"{bread_price:.2f}"`.

⧖ **8.5)** What would a fourth `f.readline()` give?

Answer: The empty string, `""`. The ▶ is at the end of the file; there is nothing left to read. `readline()` does not raise an exception at the end. It hands you nothing. So `float(f.readline())` a fourth time *would* fail, because `float("")` raises `ValueError: could not convert string to float: ''`. When you read with `readline()`, you are responsible for knowing how many lines there are.

Reading the same file twice

Here is the trap everyone falls into once. Count the lines of `sales.txt`, then try to count them again with the same `f`:

```
with open("sales.txt") as f:
    n = 0
    for line in f:
        n = n + 1
    m = 0
    for line in f:
        m = m + 1
print(n, m)
```

```
3 0
```

The second loop found nothing. ⧖ **8.6)** Why? Before reading on, picture the ▶.

Answer: The first loop read every line, so the ▶ is at the end of the file. The second loop starts where the ▶ is, finds nothing left, and never runs its body. A file does not rewind itself. To read it again, open it again with a new `with`.

When the file is not there

On the first morning there is no `sales.txt` yet. Reading it gives:

```
FileNotFoundError: [Errno 2] No such file or␣
↪directory: 'sales.txt'
```

This is an exception, like `ValueError` in chapter 6. Python cannot do the line: there is no such file. And you already know how to catch it:

```
day_total = 0
try:
    with open("sales.txt") as f:
        for line in f:
            day_total = day_total + float(line.strip())
except FileNotFoundError:
    print("No sales yet today.")
print(f"Total: {day_total:.2f}")
```

```
No sales yet today.
Total: 0.00
```

A missing file is not a disaster here. It just means nothing has been sold yet.

One common cause of `FileNotFoundError`: Python looks for the file in the folder you *ran the program from*, not necessarily the folder the program is in. If the file is plainly there and Python says it is not, check which folder your terminal is in.

Print statements: the smallest step from the sheet

Back in chapter 7 you moved the memory game onto the computer. There is an even smaller step, and you can do it inside your own program: **print what is in a cup.**

Suppose the day total comes out wrong and you want to see what each `line` really holds. Add one line:

```
with open("sales.txt") as f:
    for line in f:
        print(repr(line))
```

```
'7.50\n'
'6.00\n'
'18.90\n'
```

`repr()` shows a string *as Python sees it*: with its quotes, and with the newline written as `\n` so you can see it. Without `repr`, the newlines are hidden and you get a blank line between each one, because the line's own `\n` and `print`'s `\n` both end a line:

```
with open("sales.txt") as f:
    for line in f:
        print(line)
```

```
7.50

6.00

18.90
```

That blank line is a clue, if you know how to read it. Now you do.

> **For your glossary:** *print statement* (for checking) — a line you add only to see what a variable holds while the program runs. *repr* — shows a string with its quotes and hidden characters.

A print statement is one row of one sheet, written in code. Print three cups and you have written the Memory region yourself, and the program fills it in for you. The hard part is not the printing. It is knowing **which** cup is worth printing, and the sheets taught you that. When you have found the problem, **take the print statements out.** They are scaffolding, not part of the building.

There is one more way to see memory, the debugger. It comes at the end of chapter 10, once you can trace by hand the programs that most need it.

Tina's shop remembers

Three changes to `shop_ch07.py`.

1. Read the prices from `prices.txt` at the start, falling back to the usual prices if the file is missing. Replace the three price lines at the top with:

```
try:
    with open("prices.txt") as f:
        bread_price = float(f.readline())
        milk_price = float(f.readline())
        egg_price = float(f.readline())
except FileNotFoundError:
    print("No prices.txt; using the usual prices.")
    bread_price = 2.50
    milk_price = 1.75
    egg_price = 3.00
```

2. Write every sale to `sales.txt`, right after it is added to `day_total`:

```
            day_total = day_total + total
            sales = sales + 1
            with open("sales.txt", "a") as f:
                f.write(f"{total:.2f}\n")
```

3. At closing, read the whole day back from the file. The two lines of the summary that come from memory only know about this run, so say so. Then add the day from the file:

```
print(f"Sales this run: {sales}")
print(f"Taken in this run: {day_total:.2f}")
print(f"Left: bread {bread_stock}, milk {milk_stock},␣
↪eggs {egg_stock}")
file_total = 0.0
file_sales = 0
try:
    with open("sales.txt") as f:
        for line in f:
            file_total = file_total + float(line.
↪strip())
```

(continues on next page)

(continued from previous page)

```
            file_sales = file_sales + 1
except FileNotFoundError:
    print("No sales saved yet.")
print(f"Sales all day: {file_sales}")
print(f"Taken in all day: {file_total:.2f}")
```

Here is the whole program. Save it as `shop_ch08.py`.

```
# shop.py - version 8: prices from a file, every sale␣
↪saved to a file
shop = "Tina's Corner Shop"
try:
    with open("prices.txt") as f:
        bread_price = float(f.readline())
        milk_price = float(f.readline())
        egg_price = float(f.readline())
except FileNotFoundError:
    print("No prices.txt; using the usual prices.")
    bread_price = 2.50
    milk_price = 1.75
    egg_price = 3.00
bread_stock = 12
milk_stock = 20
egg_stock = 10
day_total = 0.0
sales = 0

print(shop)
choice = ""
while choice != "Q":
    choice = input("B)read M)ilk E)ggs Q)uit: ").
↪upper()

    # look up the item
    item = ""
    if choice == "B":
        item = "loaves of bread"
```

(continues on next page)

(continued from previous page)

```
        label = "Bread"
        price = bread_price
        stock = bread_stock
    elif choice == "M":
        item = "bottles of milk"
        label = "Milk"
        price = milk_price
        stock = milk_stock
    elif choice == "E":
        item = "cartons of eggs"
        label = "Eggs"
        price = egg_price
        stock = egg_stock

    if choice == "Q":
        print("Closing time.")
    elif item == "":
        print("Sorry, we don't sell that.")
    else:
        count = 0
        while count < 1:
            text = input(f"How many {item}? ")
            try:
                count = int(text)
                if count < 1:
                    print("Please order at least 1.")
            except ValueError:
                print("Please type a whole number,␣
↪like 3.")
        if count > stock:
            print(f"Sorry, only {stock} left.")
        else:
            total = count * price
            if count >= 10:
                total = total * 0.9
                print("10% off for 10 or more!")
            stock = stock - count
```

(continues on next page)

(continued from previous page)

```
            if choice == "B":
                bread_stock = stock
            elif choice == "M":
                milk_stock = stock
            else:
                egg_stock = stock
            day_total = day_total + total
            sales = sales + 1
            with open("sales.txt", "a") as f:
                f.write(f"{total:.2f}\n")
            for i in range(20):
                print("-", end="")
            print()
            for i in range(count):
                print(f"  {label}  {price:.2f}")
            for i in range(20):
                print("-", end="")
            print()
            print(f"  Total  {total:.2f}")

# the day's summary
print()
print(f"Sales this run: {sales}")
print(f"Taken in this run: {day_total:.2f}")
print(f"Left: bread {bread_stock}, milk {milk_stock},␣
↪eggs {egg_stock}")
file_total = 0.0
file_sales = 0
try:
    with open("sales.txt") as f:
        for line in f:
            file_total = file_total + float(line.
↪strip())
            file_sales = file_sales + 1
except FileNotFoundError:
    print("No sales saved yet.")
print(f"Sales all day: {file_sales}")
```

(continues on next page)

(continued from previous page)

```
print(f"Taken in all day: {file_total:.2f}")
```

Make `prices.txt` beside it, with the three prices. Then run the shop twice, the way Tina's day really goes.

The morning:

```
Tina's Corner Shop
B)read M)ilk E)ggs Q)uit: b
How many loaves of bread? 3
--------------------
  Bread  2.50
  Bread  2.50
  Bread  2.50
--------------------
  Total  7.50
B)read M)ilk E)ggs Q)uit: q
Closing time.

Sales this run: 1
Taken in this run: 7.50
Left: bread 9, milk 20, eggs 10
Sales all day: 1
Taken in all day: 7.50
```

The afternoon, after the computer was switched off:

```
Tina's Corner Shop
B)read M)ilk E)ggs Q)uit: m
How many bottles of milk? 2
--------------------
  Milk  1.75
  Milk  1.75
--------------------
  Total  3.50
B)read M)ilk E)ggs Q)uit: e
How many cartons of eggs? 1
--------------------
```

(continues on next page)

(continued from previous page)

```
  Eggs  3.00
--------------------
  Total  3.00
B)read M)ilk E)ggs Q)uit: q
Closing time.

Sales this run: 2
Taken in this run: 6.50
Left: bread 12, milk 18, eggs 9
Sales all day: 3
Taken in all day: 14.00
```

The memory forgot the morning. The file did not. Look at the `Left` line, too: the bread is back to 12. The stock lives in memory, so it forgot the morning as well. Exercise 10 gives the stock a file of its own.

⧖ **8.7)** Look at the banner. Which doors did the data use in this program?

Answer: In through the **Keyboard** (the menu choice and the count) and through a **File** (`prices.txt`, and `sales.txt` at closing). Out through the **Screen** (the receipt) and through a **File** (each sale written to `sales.txt`). The data **moved** from memory out to a file, and later from the file back into memory. That is the *Move* on the banner.

⧖ **8.8)** What should Tina do with `sales.txt` at the start of a new day?

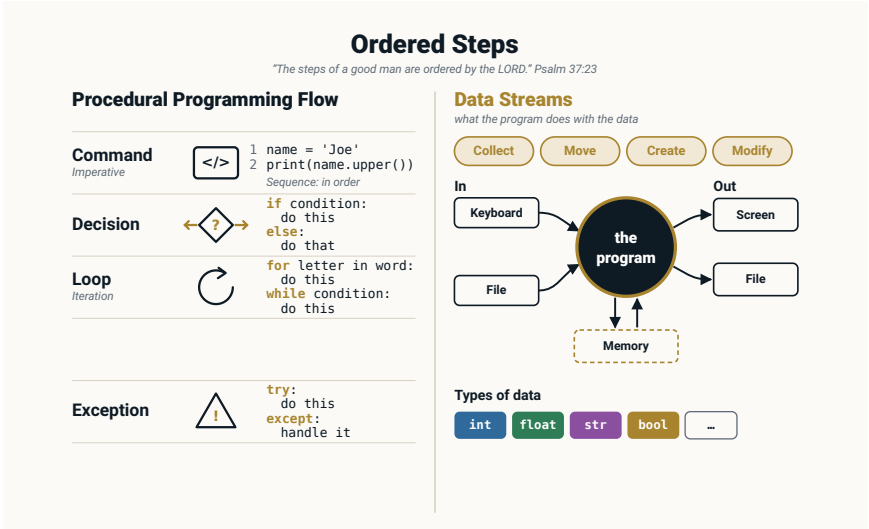
Answer: Start it fresh, or the all-day total will include yesterday. She could rename yesterday's file to keep it, or delete it. A file only remembers; it is up to you to decide what it should forget. (Exercise 10 takes this further.)

Write it down

People have always known what files are for. Long before computers, God told the prophet Habakkuk, *"Write the vision; make it plain on tablets"* (Habakkuk 2:2), so it would last and could be read later. We forget. Memory, in a computer and in us, is emptied. What is written down remains. That is also why I ask you to keep your glossary and your programs in your site, and not only in your head.

Your picture so far

Both sides of Data Streams now have their File door. Data can leave the program and come back later.



Exercises

Write every program yourself. Use print statements to check your work, then take them out. Play the memory game, with the File region, on exercise 7.

1. **Which mode?** `"r"`, `"w"` or `"a"`? (a) Add today's temperature to a file that keeps every day's temperature. (b) Read the temperatures to find the average. (c) Save a brand-new price list, replacing the old one.

2. **Write three lines.** Write a program that writes three lines to `notes.txt` with `"w"`: `Order more eggs`, `Fix the door`, `Call the baker`. Run it twice. How many lines are in the file?

3. **Now append.** Change the mode to `"a"` and run it twice more. How many lines now? Why?

4. **Number the lines.** Read `notes.txt` and print each line with its num-

ber in front: `1 Order more eggs`. Make sure there are no blank lines between them.

5. **Count the sales.** Read `sales.txt` and print how many sales it holds.
6. **The biggest sale.** Read `sales.txt` and print the largest amount in it.
7. **The memory game.** Play the five-line price reader from this chapter in `memory_game.html`, with the File region filled in. Download your sheets. On which sheet does the ▶ reach the end of the file?
8. **Check the end.** Change the price reader to read a fourth line into a cup called `extra`. If it is `""`, print `End of file: three prices, as expected.` Otherwise print `There is more in the file:` and the line.
9. **Don't trust the file.** Change the day-total reader so a line that is not a number does not stop it: print `Skipping:` and the line, and carry on. (You need a `try` *inside* the `for` loop.)
10. **Stock that lasts.** Tina's stock should not reset to 12, 20 and 10 every time the program starts. Keep it in `stock.txt`, three lines in the order bread, milk, eggs. Read it at the start (use the starting stock if the file is missing), and write it back with `"w"` when the shop closes.

Practicing with AI

> *I need help practicing reading and writing text files in Python, with open(), with, readline() and for line in f. Starting simple, give me programs that I can write. Do not use lists or split(). If I get stuck I'll ask for a hint. Then increase their complexity so I get better at designing programs that save data and read it back.*

Write every program yourself. Log the conversation as you go.

Words for your glossary

Write each of these in your own words in your site's glossary: **memory**, **file**, **open**, **close**, **file object**, **append**, **newline**, **strip**, **print statement**, **repr**.

Summary

- **Memory forgets** when the program ends. **Files remember.**
- `with open("name.txt", mode) as f:` opens the file, and closes it when the block ends.
- Modes: `"r"` read (the default), `"w"` write (**empties the file first**), `"a"` append (adds to the end).
- `f.write(text)` writes a string. Add `"\n"` yourself to end the line.
- `for line in f:` reads one line at a time. Each `line` is a string ending in `"\n"`; `line.strip()` removes it; `float()` or `int()` turns it into a number.
- `f.readline()` reads the line the ▶ is on and moves the ▶ down. At the end of the file it gives `""`.
- A file does not rewind. To read it again, open it again.
- A missing file raises `FileNotFoundError`. Catch it with `try`.
- **Print statements** (with `repr()` for strings) are the smallest step from the sheet. Take them out when you are done.
- On the banner: **File** in and **File** out, and **Move**: data going from memory to a file and back.

Answers

1. (a) `"a"`: add today's to the rest. (b) `"r"`. (c) `"w"`: replace the old list.

2. ```
   with open("notes.txt", "w") as f:
       f.write("Order more eggs\n")
       f.write("Fix the door\n")
       f.write("Call the baker\n")
   ```

   Three lines, after one run or after two. The second run emptied the file before writing.

3. **Nine.** Exercise 2 left 3 lines. Each run with `"a"` adds 3 more without emptying anything: 3 + 3 + 3 = 9.
   ```

4.
```
n = 0
with open("notes.txt") as f:
    for line in f:
        n = n + 1
        print(n, line.strip())
```

Without `strip()`, each line's own `\n` plus `print`'s would leave a blank line between them.

5.
```
sales = 0
with open("sales.txt") as f:
    for line in f:
        sales = sales + 1
print(sales, "sales")
```

6.
```
biggest = 0
with open("sales.txt") as f:
    for line in f:
        amount = float(line.strip())
        if amount > biggest:
            biggest = amount
print(f"Biggest sale: {biggest:.2f}")
```

Starting `biggest` at 0 works because no sale is negative.

7. Sheet 4, after line 4 has read the third price. The next line ▶ box holds 4, one past the last of three lines.

8.
```
with open("prices.txt") as f:
    bread_price = float(f.readline())
    milk_price = float(f.readline())
    egg_price = float(f.readline())
    extra = f.readline()
if extra == "":
    print("End of file: three prices, as expected.
↪")
else:
    print("There is more in the file:", extra.
↪strip())
```

9.
```
day_total = 0
with open("sales.txt") as f:
    for line in f:
        try:
            day_total = day_total + float(line.
↪strip())
        except ValueError:
            print("Skipping:", line.strip())
print(f"Total: {day_total:.2f}")
```

The `try` is inside the loop, so one bad line is skipped and the loop goes on to the next. If the `try` were outside the loop, the first bad line would end the whole loop.

10. At the top, in place of the three stock lines:

```
try:
    with open("stock.txt") as f:
        bread_stock = int(f.readline())
        milk_stock = int(f.readline())
        egg_stock = int(f.readline())
except FileNotFoundError:
    bread_stock = 12
    milk_stock = 20
    egg_stock = 10
```

At the very end, after the shop closes:

```
with open("stock.txt", "w") as f:
    f.write(f"{bread_stock}\n")
    f.write(f"{milk_stock}\n")
    f.write(f"{egg_stock}\n")
```

`"w"` is right here: the file should hold today's stock only, not every day's. (`int()` accepts `"9\n"`, but `strip()` does no harm if you prefer to be careful.)

Look at Tina's shop program as it stood at the end of the last chapter. It works. It sells bread, milk and eggs, it checks the stock, it gives a discount, it will not fall over when someone types `two`, and it saves every sale to a file.

Now try to find the part that reads the prices. Then find the part that asks for a whole number. You can find them, but you have to read most of the program to do it, because everything is in one long piece.

Here is a question for you before we go on.

⧖ **9.1)** The "keep asking until a whole number is typed" loop from chapter 6 is useful in many places. Tina also takes orders by phone, and wants to ask how many loaves the caller wants. How would you use that loop a second time in the same program?

Think it through before you read on.

With what you know now, you would copy it. Copy the loop, paste it, change the question. It works, but now the same lines live in two places. If you find a mistake in one, you have to remember to fix the other. A few copies later, you will forget.

Wouldn't it be nice to write those lines once, give them a name, and use the name wherever you need them? That is what this chapter is about. A **function** is a job with a name. You write the job once. After that, you say its name and the job is done.

You have been using functions since chapter 1. `print` is a function. So are `input`, `int`, `float`, `len` and `range`. Someone else wrote those jobs and gave them names, and you have been calling them by name. In this chapter you write your own.

Your first function

Here is the shop sign from chapter 1, made into a job called `show_sign`:

```
1 def show_sign():
2     print("Tina's Corner Shop")
3     print("Open 8 to 6")
4
5 show_sign()
6 print("Welcome!")
7 show_sign()
```

The screen shows:

```
Tina's Corner Shop
Open 8 to 6
Welcome!
Tina's Corner Shop
Open 8 to 6
```

Let's take it apart.

- `def` is short for *define*. Line 1 says: here is a job, and its name is `show_sign`.
- The **parentheses** `()` after the name are empty for now. Hold on to that; they are about to matter a great deal.
- The **colon** `:` and the indented lines under it are the **body** of the function: the steps of the job. You have seen this shape before with `if` and `while`. The indent is how Python knows which lines belong to the job.
- Line 5, `show_sign()`, is a **call**. A call says: go and do that job now.

⧖ **9.2)** The program has seven lines. Line 1 comes first. Why does `Tina's Corner Shop` not appear on the screen when line 1 runs?

Think it through before you read on.

Because `def` does not *do* the job. It only *describes* it. When Python reads line 1, it notes the name `show_sign` and where its lines are, then skips past the body. Nothing is printed until line 5 calls the job. It is the difference between writing a recipe on a card and cooking the meal.

Writing the recipe does not cook anything. And you can cook it as many times as you like from one card. Line 7 calls `show_sign` again, and the sign

appears again, without one extra line of code.

Jump away, do a job, come back

Go back to the banner and find **Function** on the left. Its description in chapter 0's table was: *jump away, do a job, come back.* Now you can watch that happen.

Here is the order the lines of `show_sign` actually run in. Remember, we play the memory game on the lines that *run*, not the lines as they are written.

Sheet	Line	What happens	Stack, bottom to top
1	1	Python notes the job `show_sign`. Nothing runs	1
2	5	A call: jump away to the job	5
3	2	Screen: `Tina's Corner Shop`	5, 2
4	3	Screen: `Open 8 to 6`	5, 3
5	6	The job is done. Come back. Screen: `Welcome!`	6
6	7	Another call	7
7	2	Screen: `Tina's Corner Shop`	7, 2
8	3	Screen: `Open 8 to 6`	7, 3

Look hard at the Stack column. This is the most important column in the chapter.

When the flow jumps into the job, line 5 does not leave the Stack. It stays underneath, waiting. The line on top is the line you are on. The line underneath is the line you owe a return to. That is the rule you learned with loops in chapter 5: **stack and back.** A number stays on the Stack when you have to come back to it.

With a loop, the line left waiting was the top of the loop. With a function, it is the **calling line**: the line that said `show_sign()`. That is where the flow comes back to when the job is done. When line 3 finishes, the job is over, the waiting 5 comes back to the top, line 5 is finished, and the flow moves on to line 6.

⧖ **9.3)** On sheet 7, the Stack holds 7 and 2. What does the 7 mean?

Think it through before you read on.

It means: when this job is finished, come back to line 7. The same job was called from two places, line 5 and line 7, and the Stack is how the computer remembers which one to go back to this time. The job itself does not know who called it. The Stack knows.

You did not have to learn a new rule for this. You watched a number wait on the Stack in chapter 5, while you thought you were studying loops. A function works exactly the same way.

The door: parameters

A job that always does the same thing is useful, but most jobs need to be told something. A shop assistant needs to know *what* you want to buy and *how many*. That is what the parentheses are for.

```
1 def line_total(price, count):
2     total = price * count
3     return total
4
5 bread = line_total(2.50, 3)
6 print(bread)
```

The screen shows:

```
7.5
```

In my old programming book I called the parameter list a **door**: the way data comes into a function. When you write a function, you say what will come through its door. `line_total` has a door for two things, and it gives them the names `price` and `count`. These names are the function's **parameters**.

When line 5 calls `line_total(2.50, 3)`, the values `2.50` and `3` are the **arguments**: the actual data sent through the door. The first argument goes into the first parameter and the second into the second, in order. So inside the job, `price` holds 2.50 and `count` holds 3.

⧖ **9.4)** What would `line_total(3, 2.50)` give? Is it the same?

Think it through before you read on.

It gives 7.5 too, because 3 × 2.50 is the same as 2.50 × 3. But inside the job, `price` would hold 3 and `count` would hold 2.50, which is nonsense: two and a half loaves at 3 each. The computer will not catch it. Arguments go in by **position**, first to first, second to second. Getting the order right is your job.

Sending data back: return

The door lets data in. `return` sends data out.

Line 3, `return total`, does two things at once. It ends the job, and it hands the value of `total` back to the calling line. On line 5, the call `line_total(2.50, 3)` is replaced by the value that came back, 7.5, and then the assignment happens: `bread = 7.5`.

Here is the trace. Watch the Memory and the Data Dictionary.

Shee	Line	What happens	Stack, bottom to top	Memory
1	1	Python notes the job `line_total`	1	
2	5	A call. The right side runs first, so we jump away before anything is assigned	5	
3	1	Through the door: `price` gets 2.50, `count` gets 3	5, 1	A1: 2.50, B1: 3
4	2	`total` gets 7.5	5, 2	A1: 2.50, B1: 3, C1: 7.5
5	3	`return total`: the job ends and hands back 7.5	5, 3	same
6	5	Back on line 5. `bread` gets the 7.5 that came back	5	D1: 7.5
7	6	Screen: `7.5`	6	D1: 7.5

On sheet 6, the cells `A1`, `B1` and `C1` are cleared. The job is over, and its

names went with it. Only `bread` is still alive. We come back to that in a moment, because it is one of the most important ideas in programming.

⧖ **9.5)** Here is the same function without its last line. What will the program print?

```
def line_total(price, count):
    total = price * count

bread = line_total(2.50, 3)
print(bread)
```

Think it through before you read on.

It prints:

```
None
```

The job worked out the total and then threw it away, because nothing sent it back. When a function ends without `return`, Python hands back a special value, `None`, which means *nothing came back*. If you ever see `None` where you expected a number, look for a missing `return`.

So some functions *do* something, like `show_sign`, which prints. Others *work something out and hand it back*, like `line_total`. Many do both. When you design a function, decide which it is.

Each job has its own cells

Now the important idea. Read this program and predict the screen before you run it.

```
1 def take_three(stock):
2     stock = stock - 3
3     print("Inside:", stock)
4
5 bread_stock = 12
6 take_three(bread_stock)
7 print("Outside:", bread_stock)
```

⧖ **9.6)** What are the two lines on the screen?

Think it through before you read on. Then run it.

```
Inside: 9
Outside: 12
```

Many students expect `Outside: 9`. The job took three away, didn't it? It did, but not from `bread_stock`. Let's play the game and see why.

Take out your egg carton for this one. Label one cup `bread_stock` and put 12 in it. Now the call on line 6 sends `bread_stock` through the door. What goes through a door? Not the cup. A **copy of what is in the cup**. So label a *second* cup `stock`, write 12 on a slip, and put it in. Line 2 takes the 12 out of `stock`, subtracts 3 and puts 9 back in `stock`. The `bread_stock` cup was never touched. When the job ends, the `stock` cup is emptied and put away.

Here is the same thing on the sheet, with the Data Dictionary:

Shee	Line	Stack	Data Dictionary (name, type, location, scope, pass)	Memory
1	1	1		
2	5	5	`bread_stock`, int, A1, 5:G	A1: 12
3	6	6	`bread_stock`, int, A1, 5:G	A1: 12
4	1	6, 1	`bread_stock`, int, A1, 5:G`stock`, int, B1, 1:2, val	A1: 12, B1: 12
5	2	6, 2	same two rows	A1: 12, **B1: 9**
6	3	6, 3	same two rows	A1: 12, B1: 9
7	7	7	`bread_stock`, int, A1, 5:G	A1: 12

Three columns of the Data Dictionary earn their keep here.

Location. `stock` lives in its own cell, `B1`, not in `A1`. Two names, two cells.

Pass. `val` means the parameter was passed **by value**: it got its own cell, holding a copy. Change the copy and the original never knows. In this book every parameter is passed by value, so the Pass column always says `val` for a parameter and is blank for everything else. There is another way, *by reference*,

where the parameter shares the caller's own cell. Python cannot show it with the types in this book; you will meet it in the next one.

Scope. This is where a name can be used. `5:G` means *declared on line 5, and global*: `G` stands for a name that belongs to the whole program. It lives until the program ends. `1:2` means *declared on line 1, in the call sitting at stack position 2.* Stack positions are counted from the bottom, so the program's own line is position 1, and the call to `take_three` sits at position 2. A name declared inside a call is **local** to that call. It is born when the call starts and it is gone when the call ends.

That is why `stock` is not in the dictionary on sheet 7. Only what is alive goes on the sheet.

⧖ **9.7)** Suppose you rename the parameter so that line 1 reads `def take_three(bread_stock):`, and change lines 2 and 3 to match. Does the output change?

Think it through before you read on.

No. It still prints `Inside: 9` and `Outside: 12`. Now there are two different names spelled `bread_stock`: one global, declared on line 5 (`5:G`), and one local to the call (`1:2`), each in its own cell. Same spelling, different places. The Scope column is how you tell them apart. Inside the job, the name means the local one.

Local names stay local

What happens if the calling program tries to use a name that only lives inside a job?

```
1 def sell():
2     total = 7.50
3
4 sell()
5 print(total)
```

```
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    print(total)
```

(continues on next page)

(continued from previous page)

```
          ^^^^^
NameError: name 'total' is not defined
```

`total` was born on line 2, inside the call. When the call ended, `total` went with it. On line 5 there is no such name. The error is honest: as far as line 5 is concerned, `total` has never existed.

This feels like a nuisance the first time. It is one of the best things about functions. A job cannot reach out and change the calling program's cells, and the calling program cannot reach in and change the job's. Data goes in through the door and comes out through `return`. Nothing sneaks in a window. When something goes wrong, you know the only ways in and out.

A function *can* read a global name that it never declared, but do not build on that. If a job needs something, pass it in through the door. If it works something out, send it back with `return`. Then you can read any function on its own and know everything it uses.

Design: top-down

In chapter 5 you learned to build a program by **stepwise refinement**: code a little, test it, play the memory game, refine. That is how you build *up*. Now that you can name jobs, you have a second tool for planning *down*.

Top-down design starts with the whole job in one sentence. Then you write an outline under it: the few big steps that make up the job. Then an outline under each of those, and so on, until each line at the bottom is small enough that you can see how to code it. Each of those bottom lines becomes a function.

You already know this shape. It is how you were taught to outline an essay in school: a title, the main points, the points under each main point.

Here is Tina's shop, top-down:

```
Run Tina's shop for a day
    1. Open the shop: set the stock, show the sign
    2. Keep selling until Tina quits
        2.1 Show the menu, get the choice          ->␣
```

(continues on next page)

(continued from previous page)

```
↪show_menu()
        2.2 Find what the choice is                ->␣
↪item_of(choice)
        2.3 If it is on the menu:
            2.3.1 Get how many, as a whole number ->␣
↪get_count(prompt)
            2.3.2 Find the stock of that item      ->␣
↪stock_of(...)
            2.3.3 If there is enough:
                  Sell it, print the receipt       ->␣
↪sell(choice, count)
                     Find the price                ->␣
↪price_of(choice)
                     Find the receipt's name       ->␣
↪label_of(choice)
                  Save the sale to the file        ->␣
↪save_sale(total)
                  Take the count off the stock
            2.3.4 Otherwise: say how many are left
    3. Close: show the stock left and the day's␣
↪takings -> day_total()
```

Every line with an arrow is a job small enough to write and test on its own. And notice the lines under *Sell it*: the job of selling needs smaller jobs, finding the price and the receipt's name. A function can call another function. On the Stack, that just means one more number waiting.

⧖ **9.8)** Why test each function on its own before putting it in the shop?

Think it through before you read on.

Because a small job is easy to check. If `price_of("M")` gives 1.75, you know `price_of` works, and you never have to doubt it again. If you write the whole shop and then run it for the first time, and the total is wrong, the mistake could be anywhere. That is stepwise refinement and top-down design working together: plan down, then build up, one tested job at a time.

The shop, rebuilt from named jobs

Here is the shop program built from that outline. Every line of it is something you already know from chapters 1 to 8. The only new thing is that the jobs have names.

```
# shop.py - Tina's Corner Shop, built from named jobs

def show_menu():
    return input("B)read M)ilk E)ggs Q)uit: ").upper()

def item_of(choice):
    if choice == "B":
        return "loaves of bread"
    elif choice == "M":
        return "bottles of milk"
    elif choice == "E":
        return "cartons of eggs"
    else:
        return ""

def label_of(choice):
    if choice == "B":
        return "Bread"
    elif choice == "M":
        return "Milk"
    else:
        return "Eggs"

def get_count(prompt):
    count = 0
    while count < 1:
        text = input(prompt)
        try:
            count = int(text)
            if count < 1:
                print("Please order at least 1.")
        except ValueError:
            print("Please type a whole number, like 3.
```

(continues on next page)

(continued from previous page)

```
    ↪")
34      return count
35
36  def price_of(choice):
37      try:
38          with open("prices.txt") as f:
39              bread_price = float(f.readline())
40              milk_price = float(f.readline())
41              egg_price = float(f.readline())
42      except FileNotFoundError:
43          bread_price = 2.50
44          milk_price = 1.75
45          egg_price = 3.00
46      if choice == "B":
47          return bread_price
48      elif choice == "M":
49          return milk_price
50      else:
51          return egg_price
52
53  def stock_of(choice, bread_stock, milk_stock,␣
    ↪egg_stock):
54      if choice == "B":
55          return bread_stock
56      elif choice == "M":
57          return milk_stock
58      else:
59          return egg_stock
60
61  def sell(choice, count):
62      price = price_of(choice)
63      label = label_of(choice)
64      total = count * price
65      if count >= 10:
66          total = total * 0.9
67          print("10% off for 10 or more!")
68      for i in range(20):
```

(continues on next page)

(continued from previous page)

```
        print("-", end="")
    print()
    for i in range(count):
        print(f"  {label}  {price:.2f}")
    for i in range(20):
        print("-", end="")
    print()
    print(f"  Total  {total:.2f}")
    return total

def save_sale(total):
    with open("sales.txt", "a") as f:
        f.write(f"{total:.2f}\n")

def day_total():
    total = 0.0
    try:
        with open("sales.txt") as f:
            for line in f:
                total = total + float(line.strip())
    except FileNotFoundError:
        print("No sales saved yet.")
    return total

def main():
    bread_stock = 12
    milk_stock = 20
    egg_stock = 10
    print("Tina's Corner Shop")
    choice = ""
    while choice != "Q":
        choice = show_menu()
        item = item_of(choice)
        if choice == "Q":
            print("Closing time.")
        elif item == "":
            print("Sorry, we don't sell that.")
```

(continues on next page)

(continued from previous page)

```
        else:
            count = get_count(f"How many {item}? ")
            stock = stock_of(choice, bread_stock,␣
↪milk_stock, egg_stock)
            if count > stock:
                print(f"Sorry, only {stock} left.")
            else:
                total = sell(choice, count)
                save_sale(total)
                if choice == "B":
                    bread_stock = bread_stock - count
                elif choice == "M":
                    milk_stock = milk_stock - count
                else:
                    egg_stock = egg_stock - count
    print()
    print(f"Left: bread {bread_stock}, milk
↪{milk_stock}, eggs {egg_stock}")
    print(f"Taken in all day: {day_total():.2f}")

main()
```

With `prices.txt` holding `2.50`, `1.75` and `3.00` on three lines, and no `sales.txt` yet, a short day looks like this (what the customer typed is shown after each question):

```
Tina's Corner Shop
B)read M)ilk E)ggs Q)uit: b
How many loaves of bread? 3
--------------------
  Bread  2.50
  Bread  2.50
  Bread  2.50
--------------------
  Total  7.50
B)read M)ilk E)ggs Q)uit: e
How many cartons of eggs? two
```

(continues on next page)

(continued from previous page)

```
Please type a whole number, like 3.
How many cartons of eggs? 11
Sorry, only 10 left.
B)read M)ilk E)ggs Q)uit: e
How many cartons of eggs? 10
10% off for 10 or more!
--------------------
  Eggs  3.00
  Eggs  3.00
  Eggs  3.00
  Eggs  3.00
  Eggs  3.00
  Eggs  3.00
  Eggs  3.00
  Eggs  3.00
  Eggs  3.00
  Eggs  3.00
--------------------
  Total  27.00
B)read M)ilk E)ggs Q)uit: q
Closing time.

Left: bread 9, milk 20, eggs 0
Taken in all day: 34.50
```

A few things to notice.

`main` is a job too. The top line of the outline, *run Tina's shop for a day*, became a function called `main`. The last line of the file, `main()`, is the only line outside any function that does anything. That is a common habit: the whole program is one job, made of smaller jobs.

`get_count` is the chapter 6 loop, written once. Remember question 9.1? Asking about a phone order is now one line: `phone = get_count("How many loaves for the phone order? ")`. The question comes in through the door as `prompt`; the whole number goes out through `return` when the loop is done.

`item_of` keeps chapter 5's signal. It hands back `""` for anything not on the

menu, and `main` tests `item == ""`, just as the shop has done since chapter 5.

`price_of` reads `prices.txt` every time it is called. Reading the three prices once and handing all three back would need a type this book does not use. A small file read three times a sale costs nothing you would notice. If the file is missing, `price_of` uses the usual prices.

`sell` calls `price_of`. When `price_of` is running, the Stack holds three numbers: the line in `main` that called `sell`, the line in `sell` that called `price_of`, and the line you are on.

⧖ **9.9)** Why does `main` still have that long `if`/`elif`/`else` at the bottom to take the count off the stock? Why not let `sell` do it?

Think it through before you read on.

Because of what you learned with `take_three`. If `sell` took the count off a stock parameter, it would be taking it off *its own copy*. The stock in `main` would not change. With the types in this book, a job cannot change its caller's cells; it can only send a value back. So `main`, which owns the stock, has to change it. In the next book you will learn ways to keep data and the jobs that change it together. For now, the rule is simple and safe: **in through the door, out through `return`.**

Playing the game with functions

Functions do not change the memory game. The sheet is the same sheet. Two columns just start to do more work:

- **The Stack** grows a line for every call that has not finished, with the calling line underneath. When a job ends, its line comes off and the caller is on top again, exactly where it was left.
- **The Data Dictionary** gains rows when a call starts, one for each parameter and each name declared inside, with Scope `line:position` and Pass `val` for parameters. When the call ends, those rows come off the next sheet and their cells are cleared.

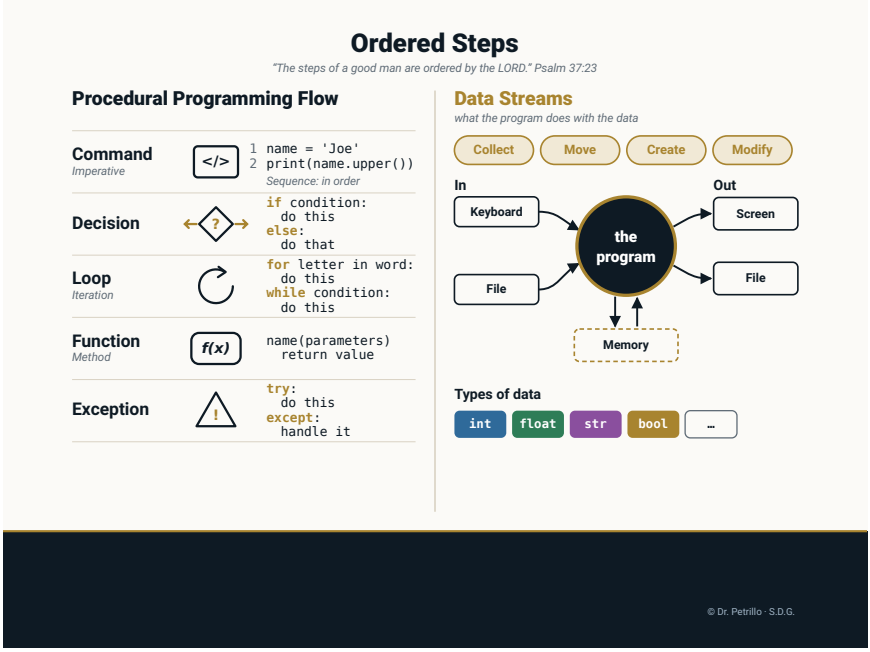
In `memory_game.html`, use **Push** with the calling line when the flow jumps into a job, and **Pop** when it comes back.

A short program with three calls can take thirty sheets. Do not skip them.

This is the chapter where students who played the game honestly start to pull ahead. The Stack you fill in here is the Stack that makes the next chapter easy.

Your picture so far

You can now find **Function** on the banner: *jump away, do a job, come back.*



When you are stuck, put your finger on the picture and ask: where am I? Am I jumping away? Am I coming back? What is waiting on the Stack?

Practicing with AI

Do the exercises below first. When you want more, ask AI like this, and log it:

> *I need help practicing functions in Python. Starting simple, give me programs to write that need functions with parameters and return values. Do not use lists. If I get stuck I'll ask for a hint. Then increase their complexity so I get better at designing programs top-down with functions.*

Exercises

Easy ones first. Write each program in your site, run it, and play the memory game on the ones marked *(sheet)*.

1. Write a function `show_hours()` that prints `Open 8 to 6`. Call it twice.
2. What does this print? Work it out before you run it.

```
def greet(name):
    print("Hello,", name)

greet("Ada")
greet("Tom")
```

3. Write a function `double(n)` that returns `n * 2`. Print `double(21)`.
4. Write a function `with_tax(amount)` that returns the amount plus 5%. Print it for 10 with two decimal places.
5. *(sheet)* Play the game on exercise 3's program. On each sheet, write the Stack, bottom to top.
6. What does this print? Explain why, using the Scope and Location columns.

```
def add_two(n):
    n = n + 2
    return n

n = 5
m = add_two(n)
print(n, m)
```

7. Write a function `is_big_order(count)` that returns `True` when the count is 10 or more and `False` otherwise. Use it in an `if`.
8. Write a function `get_yes_no(prompt)` that keeps asking until the person types `Y` or `N` (upper or lower case), then returns `"Y"` or `"N"`.
9. Top-down: write an outline for a program that sells bus tickets. A single ticket is 1.50 and a day pass is 4.00. It keeps selling until the

driver quits, and shows the day's money at the end. Mark the lines that become functions.

10. Add a function `stock_after(stock, count)` to the shop that returns the stock left after a sale. Use it in `main`. Does it let you remove the `if`/`elif`/`else` at the bottom of `main`? Explain why or why not.

Words for your glossary

Write each of these in your own words in your site's glossary: **function**, **define (`def`)**, **body**, **call**, **parameter**, **argument**, **`return`**, **`None`**, **pass by value**, **local**, **global**, **scope**, **top-down design**.

Summary

- A **function** is a job with a name. `def` describes the job; a **call** does it. Defining does not run anything.
- Function on the banner: *jump away, do a job, come back.* The **calling line** waits on the Stack: **stack and back**.
- **Parameters** are the door: data comes in, as copies, in order. **Arguments** are what you send.
- **`return`** ends the job and hands a value back. No `return` hands back `None`.
- Each call has its own cells. Names declared inside are **local** (Scope `line:position`), born when the call starts and gone when it ends. Names outside every function are **global** (`line:G`).
- In this book every parameter is passed **by value** (Pass `val`): changing it never changes the caller's variable. In through the door, out through `return`.
- **Top-down design**: an outline under an outline until each line is codeable. Each bottom line becomes a function. Plan down, build up, test each job alone.

Answers

1. ```
   def show_hours():
       print("Open 8 to 6")

   show_hours()
   show_hours()
   ```

2. ```
   Hello, Ada
   Hello, Tom
   ```

3. ```
   def double(n):
       return n * 2

   print(double(21))
   ```

   It prints `42`.

4. ```
   def with_tax(amount):
       return amount * 1.05

   print(f"{with_tax(10):.2f}")
   ```

   It prints `10.50`.

5. Line 1, Stack `1`. Line 4, Stack `4`. Line 2, Stack `4, 2`, with `n` (int, its own cell, scope `1:2`, val) holding 21. Back on line 4, Stack `4`: `print` receives the 42 that came back, and the Screen shows `42`. The row for `n` comes off the dictionary.

6. It prints `5 7`. There are two `n`s. The global `n` (scope `5:G`) holds 5 in its own cell. The parameter `n` (scope `1:2`, val) gets a copy, 5, in a different cell, and becomes 7. The function returns 7 into `m`. The global `n` was never touched.

7. ```
   def is_big_order(count):
       return count >= 10

   if is_big_order(12):
   ```

(continues on next page)
   ```

(continued from previous page)

```
    print("10% off")
```

`count >= 10` is already `True` or `False`, so you can return it directly.

8.
```
def get_yes_no(prompt):
    while True:
        answer = input(prompt).upper()
        if answer == "Y" or answer == "N":
            return answer
        print("Please type Y or N.")

print(get_yes_no("Open the shop? "))
```

`return` ends the job from inside the loop, loop and all. That is why `while True:` is safe here: the loop never has to end by itself.

9. One good outline (yours may differ):

```
Sell bus tickets for a day
    1. Start the day's money at 0
    2. Keep selling until the driver quits
        2.1 Show the choices                    ->␣
↪show_choices()
        2.2 Get the choice
        2.3 Work out the price of the choice    ->␣
↪price_of(choice)
        2.4 Add it to the day's money
    3. Show the day's money                     ->␣
↪show_total(total)
```

10.
```
def stock_after(stock, count):
    return stock - count
```

It does not remove the `if`/`elif`/`else`. `main` still has to know *which* of its three variables gets the new value:

```
if choice == "B":
    bread_stock = stock_after(bread_stock, count)
```

The job can work out the new number, but only the owner of the cell can put it there. That is pass by value.

When I was learning English, I was told never to use a word in its own definition. You may not say "a shop is a place with a shop in it." That is a good rule for dictionaries. In this chapter we are going to break it on purpose, because it turns out to be one of the most powerful ideas in programming.

A **recursive** function is a function that calls itself. **Recursion** is the last of the six structures on the banner: *call yourself, on a smaller version of the problem.*

That sounds like a trick, or like a dog chasing its tail. Many students meet recursion and go over a cliff. I want to tell you plainly why you will not. You already know everything recursion needs. You have watched a number wait on the Stack while a loop ran. You have watched a calling line wait while a job ran. Recursion is a job whose calling line happens to be inside itself. There is no second thing to learn.

Closing time

At six o'clock Tina counts down before she locks the door. Here is that countdown as a job that uses itself:

```
def countdown(n):
    if n == 0:
        print("Closed!")
    else:
        print(f"Closing in {n}...")
        countdown(n - 1)

countdown(3)
```

The screen shows:

```
Closing in 3...
Closing in 2...
```

(continues on next page)

(continued from previous page)

```
Closing in 1...
Closed!
```

Look at line 6. Inside the job `countdown`, we call `countdown`. But not with the same `n`. With `n - 1`. Each call is a slightly smaller version of the same problem: count down from 3 means *say 3, then count down from 2*. Count down from 2 means *say 2, then count down from 1*. And so on.

⧖ **10.1)** What stops it? Why doesn't it count down forever: 0, -1, -2…?

Think it through before you read on.

Line 2. When `n` reaches 0, the job prints `Closed!` and does *not* call itself again. That is the **base case**: the version of the problem so small that the answer is known without calling yourself. Every recursive function needs two parts:

- **The base case**: when to stop. Here, `n == 0`.
- **The recursive case**: do a little of the job, then call yourself on a smaller problem that is one step closer to the base case. Here, print `n`, then call with `n - 1`.

If the recursive case does not get closer to the base case, you never arrive. We will see what happens then.

Show yourself

I am not going to explain how this works. You are going to show yourself, with a pencil, and that is a different thing from being shown. It is the one that stays.

Play the memory game on the countdown. Here are the sheets, one per line that runs. The Stack is written bottom to top. The Data Dictionary shows the `n`s that are alive.

Shee	Line	Control	Stack, bottom to top	Alive in the Data Dictionary	Screen adds
1	1	Function	1		
2	8	Function	8		
3	1	Function	8, 1	`n` = 3, scope 1:2	
4	2	Decision	8, 2	`n` = 3 (1:2)	
5	5	Command	8, 5	`n` = 3 (1:2)	`Closing in 3...`
6	6	Recursion	8, 6	`n` = 3 (1:2)	
7	1	Function	8, 6, 1	`n` = 3 (1:2), `n` = 2 (1:3)	
8	2	Decision	8, 6, 2	same	
9	5	Command	8, 6, 5	same	`Closing in 2...`
10	6	Recursion	8, 6, 6	same	
11	1	Function	8, 6, 6, 1	adds `n` = 1 (1:4)	
12	2	Decision	8, 6, 6, 2	same	
13	5	Command	8, 6, 6, 5	same	`Closing in 1...`
14	6	Recursion	8, 6, 6, 6	same	
15	1	Function	8, 6, 6, 6, 1	adds `n` = 0 (1:5)	
16	2	Decision	8, 6, 6, 6, 2	same	
17	3	Command	8, 6, 6, 6, 3	same	`Closed!`
18	6	Recursion	8, 6, 6, 6	`n` = 0 is gone	
19	6	Recursion	8, 6, 6	`n` = 1 is gone	
20	6	Recursion	8, 6	`n` = 2 is gone	

Do it yourself, with the egg carton first if you need to. Every call gets **a new cup labelled `n`**. When `countdown(2)` starts, the `n` holding 3 is still there, in its own cup, waiting. You do not erase it. You put out another cup.

Now look at three things.

The Stack grows a line for every call. Each call to `countdown` leaves its line 6 waiting underneath, because each one has to be come back to. Four calls, so by sheet 17 there are four lines on top of the 8.

There are four `n`s alive at once. Same name, four different cells. How do you tell them apart? The **Scope** column. `1:2` means declared on line 1 in the call at stack position 2; `1:3` is the call above it, and so on up to `1:5`. This is why the scope has two halves. The line number alone cannot tell them apart, because all four were declared on line 1. The stack position can.

Then it unwinds. Sheets 18 to 21 are the returns. The deepest call finishes and comes off; then the one under it finds it has nothing left to do after line 6, so it finishes too; and so on, back down the Stack, in reverse order, until only the 8 is left. *Stack and back*, four times.

⧖ **10.2)** In sheets 18 to 20, nothing appears on the screen. So why bother with those sheets?

Think it through before you read on.

Because the computer really does go back through each waiting call. Here, nothing happens on the way back, so it is easy to miss. In the next example, the work happens *on the way back*, and if you skip these sheets you will not understand it.

Work on the way back

Change one thing: print *after* the call instead of before.

```
def count_up(n):
    if n > 0:
        count_up(n - 1)
        print(n)

count_up(3)
```

⧖ **10.3)** Before you run it: what does it print?

Think it through before you read on. Then run it.

```
1
2
3
```

It counts *up*, even though `n` goes down with each call. Why? Each call does its calling first, and its printing afterwards. `count_up(3)` calls `count_up(2)` before it prints anything, and that calls `count_up(1)`, which calls `count_up(0)`. That one is the base case: `n > 0` is false, so it does nothing and returns. Now the Stack unwinds. The call holding 1 comes back to line 3, goes on to line 4 and prints `1`. Then it finishes, and the call holding 2 comes back and prints `2`. Then the call holding 3.

The order of the returns decides the order of the output. If you can point to where on your own sheets that happened, you understand recursion. If you cannot, play the game again until you can.

Words: a job on a smaller word

Recursion is at home with words, because a word has a natural smaller version: the same word with one letter taken off.

First, two new ways to cut a string. You know `word[0]`, the first letter. Python can also give you a **slice**, a piece of a string:

```
word = "eggs"
print(word[0])
print(word[1:])
print(word[-1])
print(word[1:-1])
```

```
e
ggs
s
gg
```

- `word[1:]` is everything from position 1 to the end: the word without its first letter.

- `word[-1]` counts from the end: the last letter.
- `word[1:-1]` is everything between the first and the last letter.

Reverse a word

Tina's nephew likes to write the shop's signs backwards. How do you reverse a word?

```
def reverse(word):
    if len(word) <= 1:
        return word
    return reverse(word[1:]) + word[0]

print(reverse("eggs"))
```

```
sgge
```

The thinking goes like this. A word of one letter, or no letters, reversed, is itself. That is the base case. For a longer word: reverse the rest of it, then put the first letter on the end.

⧖ **10.4)** How many times is `reverse` called for `"eggs"`, and what does each call send back?

Think it through before you read on.

Four times. Here they are, going up the Stack and then coming back down:

Call	`word`	Scope of `word`	Waits for	Sends back
1st	`"eggs"`	1:2	`reverse("ggs")`	`"sgg" + "e" = "sgge"`
2nd	`"ggs"`	1:3	`reverse("gs")`	`"sg" + "g" = "sgg"`
3rd	`"gs"`	1:4	`reverse("s")`	`"s" + "g" = "sg"`
4th	`"s"`	1:5	nothing: base case	`"s"`

Read the last column from the bottom up. Each call can only finish when the one above it has sent its answer back. The answer is built on the way back down.

Does it read the same both ways?

Tina gives a small discount to anyone whose coupon code reads the same forwards and backwards, like `NOON` or `LEVEL`. A word like that is called a **palindrome**.

```
def is_palindrome(word):
    if len(word) <= 1:
        return True
    if word[0] != word[-1]:
        return False
    return is_palindrome(word[1:-1])

print(is_palindrome("NOON"))
print(is_palindrome("LEVEL"))
print(is_palindrome("BREAD"))
```

```
True
True
False
```

This one has *two* ways to stop. A word of one letter, or none, is a palindrome: return `True`. If the first and last letters differ, it is not: return `False` straight away, with no more calls. Otherwise, the answer is the same as the answer for the middle of the word.

⧖ **10.5)** How many calls does `is_palindrome("BREAD")` make?

Think it through before you read on.

One. `B` is not `D`, so the very first call returns `False`. For `LEVEL`, there are three calls: `LEVEL`, then `EVE`, then `V`, which has one letter and returns `True`. That `True` is passed back down through the other two unchanged.

When it never stops

What happens if you forget the base case?

```
def countdown(n):
    print(f"Closing in {n}...")
    countdown(n - 1)

countdown(3)
```

It prints `Closing in 3...`, `2`, `1`, `0`, `-1`, and keeps going, about a thousand times, and then the program stops with a long message that ends:

```
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

(The exact count depends on your version of Python.)

Think about the sheets. Every call puts a line on the Stack and a new `n` in memory, and none of them ever comes off, because none of them ever finishes. A real computer does not have endless memory, so Python sets a limit on how deep the Stack may grow. At the limit, it gives up. That is a **`RecursionError`**.

It is a good reminder. Humans are always limited, and what we make is limited. Only God is unlimited. A program that tries to go on forever finds its limit quickly.

When you see a `RecursionError`, ask two questions. Is there a base case? And does every call really get closer to it?

Checking your trace with the debugger

You have now played the memory game on every one of the six structures. You have also learned two other ways to see what a program is doing: the sheet, which you fill in by hand, and print statements, from chapter 8. Here is the third: the **debugger**.

A debugger is a part of your editor that can stop your program on a line you choose, let you move forward one line at a time, and show you every name and its value while it waits. Thonny and VS Code both have one.

- A **breakpoint** is the line where the program stops. In VS Code you click just left of the line number, and a red dot appears.
- **Step into** moves one line forward, and follows the flow into a function when a line calls one.
- The **Variables** panel shows the names alive right now and their values. That is your Data Dictionary and Memory, kept by the machine.
- The **Call Stack** panel shows the calls that have not finished. That is your Stack column. The debugger lists it newest first, so read it from the bottom up to match your sheet.

The buttons change from version to version. If you cannot find them, this is a fine thing to ask AI about: it is a tool, not the subject. Log it.

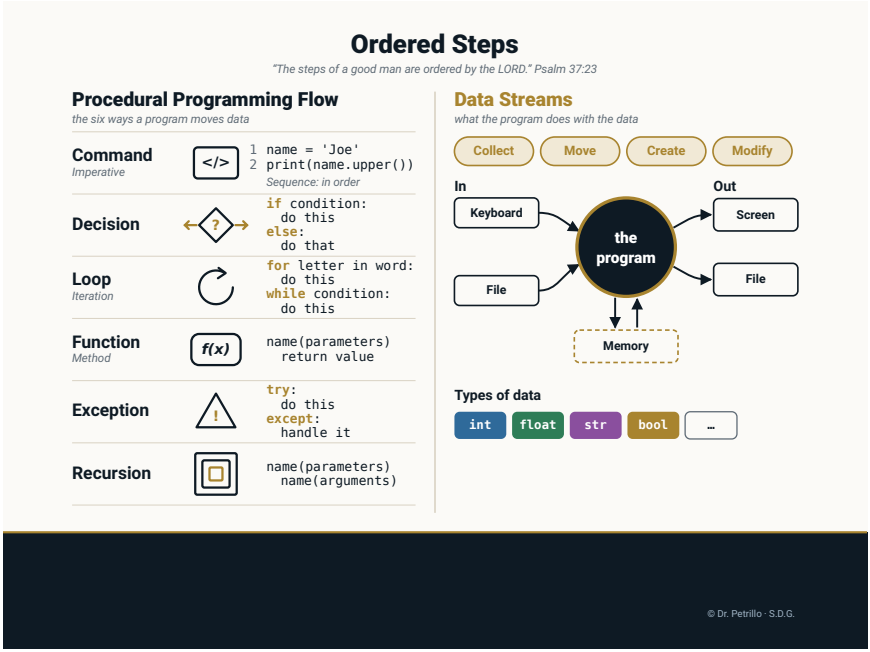
Here is the rule for this book: the debugger checks your trace. It does not replace it. Do the sheets first. Then run the same program in the debugger, step by step, and compare. Is the Call Stack the same as your Stack column, call for call? Are the values the same as your cells?

Why in that order? Because the debugger shows you memory; it does not tell you what *should* be there. A programmer who has never filled in a sheet looks at the Variables panel and sees a list of names. A programmer who has looks at it and sees whether each value is the one the line was supposed to produce. That difference is the whole skill, and the sheets are where you got it.

If you found the debugger on your own before this chapter, that is fine. Keep using it. Keep playing the game too: it is still required.

Your picture so far

Recursion is the sixth and last structure on the left of the banner: *call yourself, on a smaller version of the problem.*



All six are now on your picture. The list is closed: there is no seventh. Every program you will ever read, in any language, moves its data with these six.

Practicing with AI

Do the exercises below first. When you want more, ask AI like this, and log it:

> *I need help practicing recursion in Python. Starting simple, give me problems I can solve with a function that calls itself, using numbers and strings only, no lists. If I get stuck I'll ask for a hint. Then increase their complexity so I get better at finding the base case and the smaller problem.*

Exercises

1. What are the first and last lines that `countdown(5)` prints?
2. Here is a factorial function. (The factorial of 4 is $4 \times 3 \times 2 \times 1 = 24$.) Point to its base case. What happens if you call `factorial(0)`, and why?

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

3. Write a recursive function `shout(word, n)` that prints the word `n` times. Use no loop.
4. Write a recursive function `sum_to(n)` that returns 1 + 2 + … + n. `sum_to(4)` should return 10.
5. Type in `factorial` from exercise 2 and test it with 4.
6. *(sheet)* Play the memory game on `print(sum_to(3))`. On each sheet write the Stack, bottom to top, and every `n` alive with its scope. What does each call send back?
7. Write a recursive function `count_vowels(word)` that returns how many vowels are in the word. `count_vowels("bread")` should return 2.
8. Write a recursive function `power(base, exp)`. `power(2, 5)` should return 32.
9. Tina's receipts use a line of dashes. Write a recursive function `line_of(n)` that *returns* a string of `n` dashes. `print(line_of(10))` should print ten dashes.
10. Run your program from exercise 6 in the debugger, stepping into every call. Compare its Call Stack with your Stack column, sheet by sheet. Write one paragraph in your site: where did they match, and where (if anywhere) did your sheets go wrong?

Words for your glossary

Write each of these in your own words in your site's glossary: **recursion**, **recursive function**, **base case**, **recursive case**, **unwinding**, **slice**, **palindrome**, **`RecursionError`**, **debugger**, **breakpoint**, **step into**, **call stack**.

Summary

- A **recursive** function calls itself, on a smaller version of the problem. Recursion on the banner: *call yourself, on a smaller version of the problem.*
- It needs a **base case** (when to stop, answer known) and a **recursive case** (a little of the job, then a call that is closer to the base case).
- Each call gets its **own cells** and its own line on the **Stack**. The same name can be alive several times at once; the Scope `line:position` tells the copies apart.
- When the base case is reached, the Stack **unwinds**: each waiting call finishes in reverse order. Work done after the call happens on the way back.
- Slices: `word[1:]` (without the first letter), `word[-1]` (last letter), `word[1:-1]` (the middle).
- No base case, or no progress toward it: **`RecursionError`**.
- The **debugger** (breakpoint, step into, Variables, Call Stack) checks your hand trace. It does not replace it.

Answers

1. The first line is `Closing in 5...` and the last is `Closed!`.
2. The base case is `n == 1`. `factorial(0)` never reaches it: it calls `factorial(-1)`, then `-2`, and so on, getting further from 1 every time, until Python stops it with a `RecursionError`. To allow 0, make the base case `if n <= 1:`.
3.
```
def shout(word, n):
    if n > 0:
        print(word)
        shout(word, n - 1)

shout("Fresh bread!", 3)
```

   Here the base case is hidden: when `n` is 0, the `if` is false and the job simply ends.

4.
```
def sum_to(n):
    if n == 0:
        return 0
    return n + sum_to(n - 1)

print(sum_to(4))
```

5. `print(factorial(4))` prints `24`.

6. Four calls, with `n` = 3 (scope 1:2), 2 (1:3), 1 (1:4) and 0 (1:5). At the deepest point the Stack holds the calling line, then three waiting `return` lines, then the line you are on. On the way back, the call with 0 sends back 0, the call with 1 sends back 1 + 0 = 1, the call with 2 sends back 2 + 1 = 3, the call with 3 sends back 3 + 3 = 6, and the Screen shows `6`.

7.
```
def count_vowels(word):
    if word == "":
        return 0
    if word[0] in "aeiouAEIOU":
        return 1 + count_vowels(word[1:])
    return count_vowels(word[1:])

print(count_vowels("bread"))
```

   `word[0] in "aeiouAEIOU"` is `True` when the first letter is one of the letters in that string.

8.
```
def power(base, exp):
    if exp == 0:
        return 1
    return base * power(base, exp - 1)

print(power(2, 5))
```

9.
```
def line_of(n):
    if n == 0:
        return ""
    return "-" + line_of(n - 1)
```

(continues on next page)

(continued from previous page)

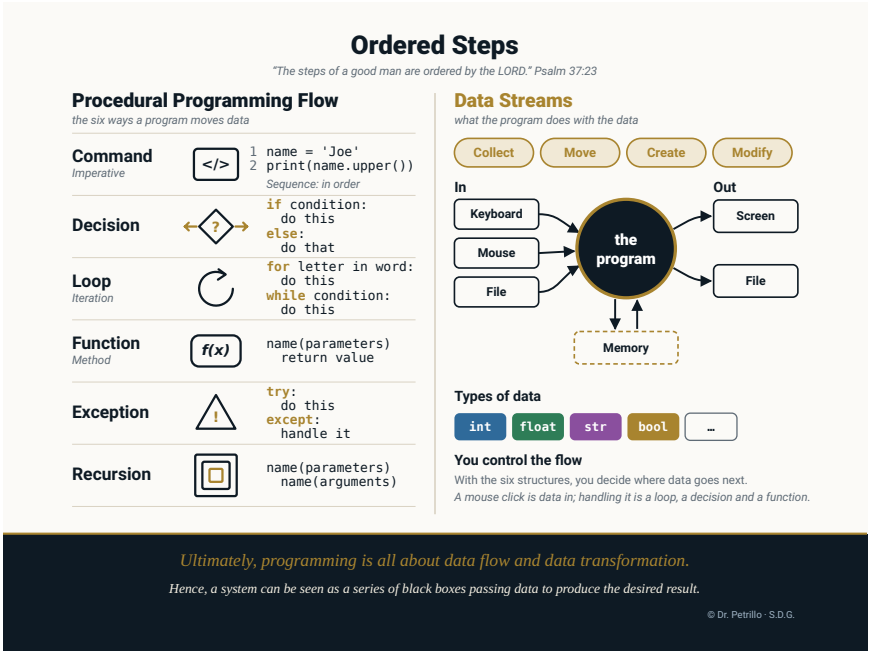
```
print(line_of(10))
```

The base case returns an empty string, `""`, and each call adds one dash on the way back.

10. Your own paragraph. A common finding: the debugger lists the newest call at the top, so it looks upside down until you read it from the bottom.

You have arrived.

At the start of this book you put a 1 in an egg cup and called it `x`. Since then, chapter by chapter, you have added one piece at a time to a picture, and each piece was something you did with your own hands before you saw it drawn. Here is the whole of it:



Nothing on it is new to you. That is the point. You did not memorise this picture. You built it.

The finished shop

Tina's shop has grown with you. Here is the last piece: before she shows the day's sales, the shop counts down to closing with the recursive `countdown` from chapter 10. Add it to the shop from chapter 9, above `main`:

```
def countdown(n):
    if n == 0:
        print("Closed!")
    else:
        print(f"Closing in {n}...")
        countdown(n - 1)
```

and call it in `main`, after the selling loop and just before the day's summary:

```
    countdown(3)
    print()
    print(f"Left: bread {bread_stock}, milk
↪{milk_stock}, eggs {egg_stock}")
    print(f"Taken in all day: {day_total():.2f}")
```

A short day now ends like this:

```
Tina's Corner Shop
B)read M)ilk E)ggs Q)uit: m
How many bottles of milk? 2
--------------------
  Milk  1.75
  Milk  1.75
--------------------
  Total  3.50
B)read M)ilk E)ggs Q)uit: q
Closing time.
Closing in 3...
Closing in 2...
Closing in 1...
Closed!

Left: bread 12, milk 18, eggs 10
Taken in all day: 3.50
```

A tour: the picture against the program

Let's walk the whole banner with the shop in hand. For every piece of the picture, there is a place in `shop.py` where it happens.

The left half: how the data moves

On the banner	In the shop
Command: do this one thing, then the next	`print("Tina's Corner Shop")`, `bread_stock = 12`, and most of the other lines. Most code is commands
Decision: ask a question, take one branch	`if count > stock:` in `main`; `elif item == "":` for a choice not on the menu; the `if`/`elif`/`else` in `price_of` that picks the price
Loop: go round again while something stays true	`while choice != "Q":` keeps the shop open; `while count < 1:` in `get_count` keeps asking; `for i in range(count):` prints the receipt; `for line in f:` adds up the day
Function: jump away, do a job, come back	every `def`: `show_menu`, `item_of`, `label_of`, `get_count`, `price_of`, `stock_of`, `sell`, `save_sale`, `day_total`, `countdown`, and `main` itself
Exception: leave this path and go handle it	`except ValueError:` when someone types `two`; `except FileNotFoundError:` when there are no sales yet, or no `prices.txt`
Recursion: call yourself, on a smaller problem	`countdown(n - 1)` inside `countdown`

All six. A small shop program uses every one of them, and so will nearly every program you ever write.

The right half: where the data comes from, sits and goes

On the banner	In the shop
In: Keyboard	`input("B)read M)ilk E)ggs Q)uit: ")` in `show_menu`, and the count in `get_count`
In: File	`prices.txt`, read line by line in `price_of`; `sales.txt`, read back in `day_total`
In: Mouse	Not in this book. But read the line under it on the banner: *a mouse click is data in; handling it is a loop, a decision and a function.* You already know all three
Out: Screen	every `print`
Out: File	`save_sale` writes each sale to `sales.txt`
Memory	every name: `choice`, `count`, `price`, `total`, the three stocks. Each sits in its own cell, and each call gets its own
Collect	taking data in: `input`, `readline`, `for line in f`
Move	carrying data from place to place: a count through a function's door, a total back through `return`, a line from the file into a cell
Create	making a value that was not there before: `total = price * count`
Modify	changing a value in place: `bread_stock = bread_stock - count`, `total = total * 0.9`
Types of data	`int` for the counts and stocks, `float` for prices and totals, `str` for the choice and the file's lines, `bool` for every question an `if` or `while` asks. The … is every type beyond this book

The bottom line

> *Ultimately, programming is all about data flow and data transformation. Hence, a system can be seen as a series of black boxes passing data to produce the desired result.*

Now you can see what that second sentence means. A **black box** is something you use by what goes in and what comes out, without looking inside. Every function you wrote is one. `price_of` takes a choice in through its door and sends a price out through `return`. `main` does not need to know whether the price came from a file, a calculation or a fixed number. If `price_of` changes inside, nothing around it breaks. The shop is a series of black boxes, passing data to produce a receipt and a day's total.

And you can open every one of those boxes and see exactly what is inside, down to the last cell. That is what the memory game gave you.

⌛ **11.1)** Pick any line of `shop.py`. Can you name which of the six structures it is, and which part of the right half of the banner its data belongs to?

Think it through before you read on.

You should be able to, for every line. If there is a line where you cannot, that line is your next memory game.

Where are you on the picture?

For the whole of this course, when you were stuck, your teacher could put a finger on this picture and ask you one question: *where are you on it?* More often than not, you found the answer yourself, and nobody told you anything.

That question works because the picture is **closed**. There are six ways a program moves data. Not six for beginners and more later: six. Wherever you are stuck, you are somewhere on this page.

Now the question is yours. Here is how to ask it of yourself.

1. **Find the first thing that is wrong.** Not where the program crashed, or where you noticed. The first line where the screen, a file or a cell holds something you did not expect.
2. **Put your finger on the left half.** Which of the six is that line? Is the flow going where you meant it to go?
3. **Put your finger on the right half.** Where did the data in that line come from? Where does it sit? Where is it going?
4. **Play the game from the last line you trust.** One sheet per line that runs, until the sheet and your expectation disagree. There is your mistake.

Here is a real one. Tina sells exactly 10 cartons of eggs and the discount does not appear. You look at `sell` and find:

```
    if count > 10:
```

Where are you? On **Decision**: a question that sent the flow down the wrong branch. Ten is not more than ten. The question should be `count >= 10`. You did not need a teacher. You needed the picture and the sheet.

⧖ **11.2)** Here is a second one. Tina runs the shop on Monday and sells 7.50 of bread. On Tuesday she sells 3.50 of milk, and at closing the shop says *Taken in all day: 11.00*. Where are you on the picture?

Think it through before you read on.

On **Out: File**, and then **In: File**. `save_sale` opens `sales.txt` with `"a"`, which adds to the end of the file. Nothing ever empties it, so Tuesday's total includes Monday's sale. The data flowed exactly where the program sent it. The program sent it to the wrong place for what Tina wanted. Exercise 2 asks you to fix it.

The major project

Now it is your turn to build something whole. In the old days, students in my courses ended a book by writing something of their own that used everything in it, and I still think it is the best way to finish. It is where you find out that you can.

What to build

Your own program, chosen by you and approved by your teacher. Pick something you know from life, the way Tina's shop came from a shop. Some ideas:

- a ticket seller for a bus, a cinema or a football match;
- a school tuck shop, a café or a market stall;
- a quiz that reads its questions and answers from a file, one line each, and keeps your score;
- a savings tracker that records each deposit in a file and shows the balance;

- a parking meter that charges by the hour;
- a word game: reversed names, palindromes, counting letters.

Requirements

Your program must:

1. Use **all six structures**, each for a real reason. Recursion counts only if it does a real part of the job.
2. Take data from the **keyboard**, show results on the **screen**, and both **read** and **write** a **file**.
3. Use the base types (`int`, `float`, `str`, `bool`) and strings. No lists, as in the rest of this book.
4. Have at least **four functions of your own**, each taking data in through its parameters and sending results out with `return` where it works something out.
5. Not crash on bad input. A letter where a number belongs is handled.
6. Run from start to finish without help.

How to build it

1. **Propose it.** One paragraph in your site: what it does, who it is for, and what goes in and comes out. Get your teacher's approval before you write code.
2. **Design it top-down.** An outline in your site, an outline under each point, until each line at the bottom is a job you can code. Mark the lines that become functions.
3. **Build it by stepwise refinement.** Code one job, test it alone, play the game on it, then add the next. Save each working version in your site (`project_v01.py`, `project_v02.py`, …) so you, and your teacher, can see how it grew.
4. **Prove it with the memory game.** In `memory_game.html`, play and download the sheets for at least three parts: a function call with a parameter and a `return`; the recursive part, all the way down and back; and a file being read, with the ▶ moving. Put the downloaded files in your site.

5. **Explain it against the picture.** A page in your site that walks your program the way this chapter walked the shop: for each piece of the banner, where it happens in your program. Write it so a friend across the kitchen table could follow it.
6. **Keep your AI log.** Everything in "Using AI in This Course" still holds. AI may look things up and give you practice. It may not write, fix or design your program.

This project is yours. Do it without help from other people, and do not copy code from the internet or anywhere else. This is the only way you'll learn to *write* code, rather than only being able to change someone else's.

How it is assessed

Part	What your teacher looks for	Weight
Design	A clear top-down outline whose functions match the program	15
It works	Runs start to finish; bad input handled; files read and written	20
The six and the file	All six structures used for real reasons; keyboard, screen and file	15
Memory game proof	Sheets for a call, the recursion and a file read, correct line by line	20
Explanation	The program walked against the picture, plainly, in your own words	15
Readable code	Clear names; comments where something is hard, not where it is obvious	5
AI log	Honest and complete	10

Your teacher may change the weights. And be ready to show your program to the class and answer their questions. If you can explain it, it is yours.

Practicing with AI

For the project you are on your own. For more practice before you start, ask AI like this, and log it:

I have learned the six procedural structures in Python: command, decision, loop, function, exception and recursion, plus keyboard, screen and file input and output, with no lists. Starting simple, give me small whole programs to design and write that use several of them together. If I get stuck I'll ask for a hint. Then increase their complexity so I get better at designing whole programs top-down.

Exercises

1. On blank paper, with the book closed, draw the whole banner from memory. Then compare it with the picture. What did you leave out? Play the memory game on a program that uses that piece.
2. Fix Tina's Monday-and-Tuesday problem from question 11.2: at the start of each day, the shop should begin with an empty `sales.txt`. Which part of the picture did you change?
3. A student's shop prints `Sorry, only 12 left.` when someone asks for 3 loaves at the start of the day. Their `main` has `if count < stock:` where it should have `if count > stock:`. Using the four steps in "Where are you on the picture?", write how you would find this mistake without being told.
4. Choose one of your own functions from an earlier chapter and describe it as a black box: what goes in through the door, what comes out, and what the caller does not need to know.
5. Write your major project proposal (step 1 of "How to build it").

Words for your glossary

Write each of these in your own words in your site's glossary: **black box**, **closed (map)**, **major project**.

Summary

- The banner is complete, and every piece of it is something you did before you saw it drawn.
- **Left half**: the six ways a program moves data, a closed list. **Right half**: where data comes from (keyboard, file), where it sits (memory, in types) and where it goes (screen, file), and what the program does with it (collect, move, create, modify).
- A program is a series of **black boxes** passing data. Every function is one: data in through the door, out through `return`.
- **Where are you on the picture?** Find the first thing that is wrong; name its structure; follow its data; play the game from the last line you trust.
- The **major project**: your own program, all six structures and a file, designed top-down, built by stepwise refinement, proved with the memory game, explained against the picture.

Answers

1. Your own drawing. Students most often leave out the Stack's part in Loop and Function, the Collect/Move/Create/Modify row, or Exception.
2. Add a few lines at the start of `main`:

```
    new_day = input("Start a new day? Y/N: ").
↪upper()
    if new_day == "Y":
        with open("sales.txt", "w") as f:
            pass
```

Opening a file with `"w"` empties it, and `pass` means *do nothing else*. Why ask first? Remember chapter 8: Tina closes for lunch and runs the shop again in the afternoon. If the program emptied the file every time it started, the afternoon would wipe out the morning's sales. So only Tina knows when a new day begins, and the program asks her. You changed two pieces of the picture: a **Decision** (is this a new day?) and **Out: File** (empty the file once, then add to it with `"a"` for each sale).

3. The first wrong thing: the screen says `Sorry` when there are 12 loaves and the customer wants 3. That line comes from a **Decision**, so the question is sending the flow down the wrong branch. On the right half, the data is `count` (3, from the keyboard) and `stock` (12, from memory). On the sheet for the `if` line, `3 < 12` is `True`, so the flow takes the `Sorry` branch. The question is backwards: it should ask whether the count is *more* than the stock.

4. For example: `get_count(prompt)`. In through the door: the question to ask. Out through `return`: a whole number, 1 or more. The caller does not need to know about the loop, the `try`, or how many times the person typed something wrong.

5. Your own proposal.

What comes next

In your next course you will meet **object-oriented programming**, and a new language, C++. You will learn to bundle data together with the jobs that change it, and the "by reference" you met in the Pass column will finally be used. It will look new. It is not. Inside every object is a method, and inside every method are these same six structures, moving data through this same memory. When something there stops making sense, come back down to this picture, and to the sheets. They will still be true.

"The steps of a good man are ordered by the LORD" (Psalm 37:23). A program is ordered steps. So, I pray, is a good life. Thank you for walking these with me.

Look here first when you forget how to write something. If the card does not have it, ask AI, starting with: *"Answer only with a template that uses blanks, like the ones on my syntax card. Do not write code for my problem."* Log it as you ask.

How to read the card. Python's own words are written as they must be typed: `if`, `while`, `print`. Plain words such as *condition*, *name* and *do this* are **blanks**: you fill them in. Indented lines (four spaces) belong to the line above them. The number in brackets is the chapter that teaches it.

The six structures

Command (1)

One thing, then the next line.

```
name = value
print(something)
# a comment: Python skips it, people read it
```

Decision (4)

```
if condition:
    do this
elif other condition:
    do this instead
else:
    do that
```

`elif` and `else` are optional. Use as many `elif` as you need.

Loop (5, 7)

```
while condition:
    do this
```

```
for letter in word:
    do this with letter
```

```
for counter in range(how many):
    do this
```

`range(stop)` counts 0, 1, … up to one less than *stop*. `range(start, stop)` starts at *start*. `range(start, stop, step)` counts by *step*.

Exception (6)

```
try:
    do this
except ErrorName:
    handle it
```

Keep asking until the answer is good (Python's problems are exceptions; your rules are decisions):

```
count = 0
while count < 1:
    text = input("prompt")
    try:
        count = int(text)
        if count < 1:
            print("message for your rule")
    except ValueError:
        print("message for a word that is not a number
↪")
```

Start with a value that fails the test, so the loop runs at least once.

Function (9)

```
def name(parameters):
    do this
    return value
```

Calling it:

```
answer = name(arguments)
```

Data comes in through the door (the parameters, as copies) and goes out through `return`. Names made inside a function are local: they are gone when it ends.

Recursion (10)

```
def name(parameters):
    if base case:
        return answer
    return name(smaller arguments)
```

or, for a job that does something instead of handing a value back:

```
def name(n):
    if n == 0:
        do the last thing
    else:
        do this
        name(n - 1)
```

Every recursive function needs a base case, a case it answers without calling itself, and every call must get closer to it.

Output and input

You want	Write
Show text (1)	`print("text")`
Show a blank line (1)	`print()`
Show several things, spaces between (1)	`print(this, that)`
Stay on the same line (7)	`print(something, end="")`
Show values inside text (3)	`print(f"Total: {total}")`
A number with 2 decimal places (3)	`f"{total:.2f}"`
Read typed text (3)	`name = input("prompt")`
Read a whole number (3)	`count = int(input("prompt"))`
Read a decimal number (3)	`price = float(input("prompt"))`

`input` always gives a string. Turn it into a number with `int()` or `float()`.

To check what a variable holds while the program runs (8): `print(repr(name))`. Take it out when you are done.

Comparing and combining (4)

Means	Write
equal	`==`
not equal	`!=`
less, less or equal	`<`, `<=`
greater, greater or equal	`>`, `>=`
both true	`condition and condition`
at least one true	`condition or condition`
the opposite	`not condition`
one of these letters (7)	`letter in "aeiou"`

= puts a value in a variable. == asks whether two values are equal.

Arithmetic (2)

Means	Write	Example	Gives
add, subtract	`+`, `-`	`12 - 3`	`9`
multiply	`*`	`2.50 * 3`	`7.5`
divide	`/`	`7 / 2`	`3.5`
whole part of a division	`//`	`125 // 12`	`10`
remainder	`%`	`125 % 12`	`5`
power	`**`	`2 ** 3`	`8`

`*` and `/` come before `+` and `-`. Brackets come first of all.

Add to what is already there: `total = total + amount`.

The four base types (2)

Type	Holds	Examples	Turn a value into it
`int`	whole numbers	`12`, `-3`	`int(value)`
`float`	decimal numbers	`2.50`, `0.5`	`float(value)`
`str`	text	`"Bread"`	`str(value)`
`bool`	true or false	`True`, `False`	the answer to a question, such as `count >= 10`

`type(value)` tells you the type.

Strings (2, 7, 10)

You want	Write
Join two strings	`text + more_text`
How many characters (7)	`len(word)`
The first character (7)	`word[0]`
The last character (7)	`word[len(word) - 1]`, or (10) `word[-1]`
All but the first character (10)	`word[1:]`
All but the first and last (10)	`word[1:-1]`
Capitals, small letters (4, 7)	`word.upper()`, `word.lower()`
Remove spaces and the newline at the ends (8)	`line.strip()`

Characters are counted from 0.

Files (8)

Add a line to the end of a file (makes the file if it is not there):

```
with open("name.txt", "a") as f:
    f.write(f"{total:.2f}\n")
```

Start a file fresh (throws away what was in it):

```
with open("name.txt", "w") as f:
    f.write(text + "\n")
```

Read one line at a time:

```
with open("name.txt") as f:
    first = f.readline()
    second = f.readline()
```

Read every line:

```
with open("name.txt") as f:
    for line in f:
        do this with line
```

Each line read from a file ends with `"\n"`. `line.strip()` takes it off; then `float()` or `int()` turns the text into a number. At the end of the file, `f.readline()` gives `""`. A file does not rewind: to read it again, open it again.

A missing file raises `FileNotFoundError`. Catch it with `try`:

```
try:
    with open("name.txt") as f:
        do this
except FileNotFoundError:
    handle it
```

Errors you will meet

Error	Usually means
`SyntaxError`	A line breaks Python's rules: a missing quote, bracket or colon. Nothing runs.
`IndentationError`	Spaces at the start of a line where they do not belong, or missing where they do.
`NameError`	A name Python does not know: a misspelling, a wrong capital, or a variable used before it is given a value.
`TypeError`	The wrong type for the job, such as joining a string to a number.
`ValueError`	The right type but a value that will not work, such as `int("two")`.
`IndexError`	A character past the end of a string, such as `word[4]` when `word` is `"Tina"`.
`ZeroDivisionError`	Dividing by zero.
`FileNotFoundError`	The file is not where the program looked.
`RecursionError`	A function called itself too many times; check the base case.

Ordered Steps

AI can write a program in seconds. So why learn to write one yourself?

Because you cannot check what you do not understand. The people who will guide AI well, and catch it when it is wrong, are the people who know what a program does, step by step, to the memory inside the machine.

This book teaches that, from the very beginning. You start with an egg carton and slips of paper, being the computer yourself. Then, one piece at a time, you write a real program in Python for Tina's corner shop: a sign, prices, orders, a till that keeps selling, mistakes caught, the day's sales kept in a file, and every job given its own name. With each piece the picture of how a program works grows, until at the end you see the whole of it and know where every part came from.

Along the way you play the memory game: tracing every line by hand, on paper, then on a spreadsheet, then on screen, until you can see what the computer sees. And you learn to use AI wisely: when it builds you up, and when it would do your growing for you.

About the Author

Dr. Anthony J. Petrillo, II has taught computer science for more than forty years, from a technical college he founded in Jos, Nigeria, to university classrooms in the United States. He holds a Ph.D. in mathematics education from the State University of New York at Buffalo, and has written programming textbooks since 1992. To his grandchildren and his students he is Baba Computer.