

# Python

From Scratch to text: functions, variables, conditionals, loops, and what can go wrong with numbers. Follows CS50x Lecture 1, with every example rewritten in Python.

## Contents

|                           |                      |                               |
|---------------------------|----------------------|-------------------------------|
| 01 Welcome                | 08 The terminal      | 15 Loops and meow.py          |
| 02 Source code            | 09 Conditionals      | 16 Functions                  |
| 03 Editor and terminal    | 10 Types             | 17 Correctness, design, style |
| 04 Hello, world           | 11 Formatting output | 18 Mario                      |
| 05 From Scratch to Python | 12 Variables         | 19 Operators                  |
| 06 Libraries and docs     | 13 compare.py        | 20 Summing up                 |
| 07 Hello, you             | 14 agree.py          |                               |

## 01 Welcome

Last time we programmed in Scratch, a visual language where you snap blocks together. Today we write the same ideas as text, in Python.

Learning to program can feel like drinking from a fire hose. That is normal. What matters is how far you have come in a few weeks, not how fast you understood each slide.

Every idea from Scratch carries over. **Functions**, **conditionals**, **loops** and **variables** are the building blocks of every programming language. Only the way we write them changes.

## 02 Source code

A computer only understands binary. The patterns of 0s and 1s it runs are called **machine code**. People write **source code**: instructions in a language people can read.

Something must turn one into the other. Languages like C use a **compiler**, which translates the whole file into machine code before you run it. Python uses an **interpreter**, which reads your source code and runs it straight away.

source code → **interpreter** → it runs

(Strictly, Python first turns your file into an in-between form called bytecode, but you never see that step.) The trade-off: interpreted code runs a little slower, but you see results faster, which is what matters when you are learning.

Today you learn to write code that works, and also how to write code that is **good**.

## 03 Editor and terminal

You need two tools, side by side:

- A **code editor**, such as VS Code. It has a file explorer on the left and a text editor in the middle. It is a **graphical user interface** (GUI): windows, menus and a mouse.
- A **terminal**, usually at the bottom of the editor. It is a **command-line interface** (CLI): you type commands and press Enter.

Write in the editor, run in the terminal. Check that Python is installed by typing `python --version`. On some computers the command is `python3`.

---

## 04 Hello, world

Two commands write and run your first program:

```
$ code hello.py
$ python hello.py
```

`code hello.py` creates the file and opens it in the editor. Use lowercase file names and end them in `.py`. Type this:

```
hello.py

# A program that says hello to the world

print("hello, world")
```

Save, then run `python hello.py` in the terminal. You see `hello, world`.

The line starting with `#` is a **comment**. Python ignores it; it is a note for people. Every other character matters: if you mistype one, the program will not run.

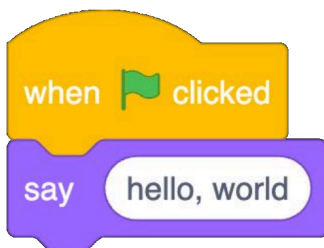
Look in the file explorer. There is only `hello.py`. Python ran your source code directly, so there is no separate machine-code file.

In C the same program needs `#include <stdio.h>`, a `main` function, braces, a semicolon and `\n`. You must compile with `make hello` and then run `./hello`, which is a second file holding machine code.

---

## 05 From Scratch to Python

Scratch's **say** block shows text on the screen. Python's `print` function does the same.



```
print("hello, world")
```

We **call** the function `print`. The value in the parentheses, `"hello, world"`, is its **argument**. Text in quotes is called a **string**. Single and double quotes both work; pick one and stay with it.

Mistakes are normal. Delete the closing quote and run the program again:

```
$ python hello.py
File "hello.py", line 3
    print("hello, world)
      ^
SyntaxError: unterminated string literal
```

Read the error: it gives the line number and points at the problem. Python is also case-sensitive, so `Print("hi")` gives `NameError: name 'Print' is not defined`.

`print` adds a new line at the end for you. To stop it, pass `end=""`. Inside a string, a backslash starts an **escape sequence**:

| Write                            | You get                                      |
|----------------------------------|----------------------------------------------|
| <code>\n</code>                  | a new line                                   |
| <code>\"</code>                  | a double quote inside a double-quoted string |
| <code>\'</code>                  | a single quote inside a single-quoted string |
| <code>\\</code>                  | one backslash                                |
| <code>print("hi", end="")</code> | no new line after <code>hi</code>            |

## 06 Libraries and docs

A **library** is code that other people wrote for you to use. `print` and `input` are **built in**, so you can use them straight away. Other tools live in libraries that you **import** first:

```
import math

print(math.sqrt(16))
```

To learn what a function does, read the documentation at `docs.python.org`, or type `help(print)` at the Python prompt (run `python` with no file name; leave with `exit()`).

In C CS50 has to give students its own library, `cs50.h`, with `get_string`, `get_int` and friends, because C has no easy way to read input. Python's `input` does that job.

## 07 Hello, you

In Scratch you could ask "What's your name?" and say hello with the answer joined on. Here is a first try in Python:



```
hello.py

# input, and print with the wrong text

answer = input("What's your name? ")
print("hello, answer")
```

It prints `hello, answer`, the word itself. Text inside quotes is printed exactly as written. We want the **value** stored in `answer`:

```
hello.py
```

```
# input, and print with an f-string

answer = input("What's your name? ")
print(f"hello, {answer}")
```

```
$ python hello.py
What's your name? David
hello, David
```

- `input` shows a prompt, waits for the user and then **returns** what they typed, always as a string. That result is its **return value**.
- `answer` is a **variable**: a named place to keep a value. The `=` sign means **assignment**: copy the value on the right into the name on the left. Read it right to left.
- The `f` before the quotes makes a **format string**. Python fills in each `{ }` with the value of what is inside.

Two other ways also work: `print("hello, " + answer)` joins two strings with `+`, and `print("hello,", answer)` passes two arguments, which `print` separates with a space.

---

## 08 The terminal

The command line is often faster than the GUI for working with files. Common commands:

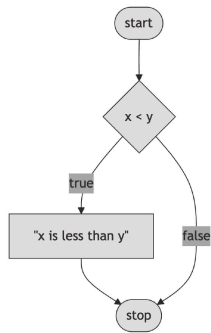
| Command            | What it does                            |
|--------------------|-----------------------------------------|
| <code>cd</code>    | change the current directory (folder)   |
| <code>cp</code>    | copy files and directories              |
| <code>ls</code>    | list the files in the current directory |
| <code>mkdir</code> | make a directory                        |
| <code>mv</code>    | move or rename files and directories    |
| <code>rm</code>    | remove (delete) files                   |
| <code>rmdir</code> | remove (delete) empty directories       |

The one you will use most is `ls`. Type it now to see the files in your current folder.

---

## 09 Conditionals

Scratch's **if** and **if-else** blocks let a program decide. Python writes them with a colon and an indented block:



# Conditionals that are mutually exclusive

```

if x < y:
    print("x is less than y")
else:
    print("x is not less than y")
  
```

The **indentation is the syntax**. The four spaces show which lines belong to the `if`, just as the C-shaped Scratch block wraps around the blocks inside it. A wrong indent gives an `IndentationError`.

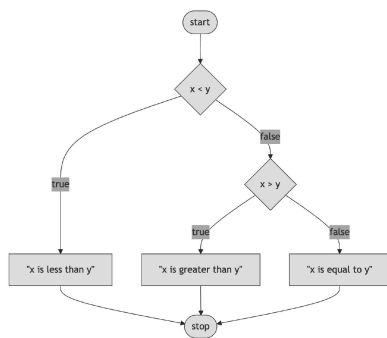
We can plan for three outcomes. `elif` is short for "else if":

# Conditional that isn't necessary

```

if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
elif x == y:
    print("x is equal to y")
  
```

Is the last question needed? If `x` is not less than `y` and not greater, it can only be equal. We can drop that question:



# Compare integers

```

if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x is equal to y")
  
```

Both versions are correct, but the second asks fewer questions. That is better **design**.

## 10 Types

Every value has a **type**. Python works out the type for you, but the types are still there, and they decide what operators do.

| Type               | Example              | Kind of value                           | Similar types in C                       |
|--------------------|----------------------|-----------------------------------------|------------------------------------------|
| <code>int</code>   | <code>42</code>      | a whole number                          | <code>int</code> , <code>long</code>     |
| <code>float</code> | <code>3.14</code>    | a number with a decimal point           | <code>float</code> , <code>double</code> |
| <code>str</code>   | <code>"hello"</code> | text, of any length, even one letter    | <code>string</code> , <code>char</code>  |
| <code>bool</code>  | <code>True</code>    | <code>True</code> or <code>False</code> | <code>bool</code>                        |

Ask Python for a value's type with `type` : `type(42)` gives `<class 'int'>` , and `type("42")` gives `<class 'str'>` . The quotes make it a string.

## 11 Formatting output

The `{ }` in an f-string can hold any value or expression. After a colon you can say how to show it:

| Write                         | Shows                  | Meaning                       |
|-------------------------------|------------------------|-------------------------------|
| <code>f"{name}"</code>        | <code>David</code>     | the value as it is            |
| <code>f"{x + y}"</code>       | <code>3</code>         | the result of an expression   |
| <code>f"{3.14159:.2f}"</code> | <code>3.14</code>      | a float with 2 decimal places |
| <code>f"{1000000: ,}"</code>  | <code>1,000,000</code> | a number with commas          |

In C `printf` uses **format codes** instead, one per type: `%s` for a string, `%i` for an int, `%li` for a long, `%f` for a float and `%c` for a char. An f-string works out the type for you.

## 12 Variables

Make a variable called `counter` and give it the value 0:

```
counter = 0
```

Add one to it. Read this right to left: take `counter` , add 1, and store the result back in `counter` .

```
counter = counter + 1
```

The short form does the same thing, and `--` subtracts:

```
counter += 1
counter -= 1
```

In C you must state the type, as in `int counter = 0;` , and you can write `counter++` and `counter--` . Python has no `++` or `--` .

## 13 compare.py

Put this together into a program. Type `code compare.py` :

compare.py

```
# Conditional, Boolean expression, relational operator

# Prompt user for integers
x = int(input("What's x? "))
y = int(input("What's y? "))

# Compare integers
if x < y:
    print("x is less than y")
```

`input` always returns a string. `int(...)` converts that string to an integer.

**Watch out** Leave out `int` and type 2 and 10. The program says 2 is not less than 10, because it compares the strings "2" and "10" letter by letter, and "2" comes after "1". Convert input to a number before you compare it.

Now handle every outcome:

compare.py

```
# Conditionals

# Prompt user for integers
x = int(input("What's x? "))
y = int(input("What's y? "))

# Compare integers
if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x is equal to y")
```

Run it with `python compare.py` and test all three outcomes. Almost any piece of code can be drawn as a flowchart like the ones above. Drawing it is a good way to see whether the design is simple.

The comparison operators:

| Operator                             | Question                                |
|--------------------------------------|-----------------------------------------|
| <code>==</code> <code>!=</code>      | equal to? not equal to?                 |
| <code>&lt;</code> <code>&lt;=</code> | less than? less than or equal to?       |
| <code>&gt;</code> <code>&gt;=</code> | greater than? greater than or equal to? |

## 14 agree.py

Ask the user a yes-or-no question. Type `code agree.py` :

agree.py

```
# Comparing against lowercase letters

# Prompt user to agree
s = input("Do you agree? ")

# Check whether agreed
if s == "y":
    print("Agreed.")
elif s == "n":
    print("Not agreed.")
```

Note `==`, two equal signs, which asks "are these equal?". A single `=` would assign. What if the user types a capital `Y`? We could add more questions:

agree.py

```
# Comparing against lowercase and uppercase letters

s = input("Do you agree? ")

if s == "y":
    print("Agreed.")
elif s == "Y":
    print("Agreed.")
else:
    print("Not agreed.")
```

It works, but the same line appears twice. Combine the two questions with `or`:

agree.py

```
# Logical operators

s = input("Do you agree? ")

if s == "Y" or s == "y":
    print("Agreed.")
else:
    print("Not agreed.")
```

Python's logical operators are English words: `and`, `or`, `not`. Strings also have their own functions, called **methods**. `s.lower()` gives a lowercase copy of `s`, so you can accept `y`, `Y`, `yes` and `YES` at once:

agree.py

```
# Lowercase first, then check against a list

s = input("Do you agree? ").lower()

if s in ["y", "yes"]:
    print("Agreed.")
else:
    print("Not agreed.")
```

In C a single letter has its own type, `char`, written with single quotes ( `'y'` ), and "or" is written `||`. Python has no separate type for one letter: `"y"` is a string of length 1.

## 15 Loops and meow.py

Scratch's **repeat** and **forever** blocks become loops. Type `code meow.py`:

meow.py

```
# Opportunity for better design

print("meow")
print("meow")
print("meow")
```

It works, but the same line is repeated. A `while` loop repeats its block for as long as its condition is true:

meow.py

```
# Better design

i = 3
while i > 0:
    print("meow")
    i -= 1
```

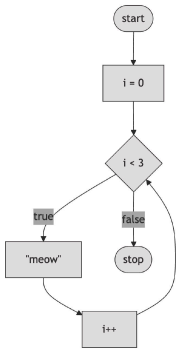
`i` starts at 3. Each time round, the cat meows and `i` goes down by 1: 3, 2, 1, then 0, when `i > 0` is false and the loop stops. We can count up instead:

meow.py

```
# Count up from 1

i = 1
while i <= 3:
    print("meow")
    i += 1
```

Programmers usually count from zero. You will see why when we meet lists.



```
meow.py

# Better design: count from 0

i = 0
while i < 3:
    print("meow")
    i += 1
```

When you know how many times to repeat, a `for` loop is simpler. It matches Scratch's **repeat** block:



```
meow.py

# Better design

for i in range(3):
    print("meow")
```

`range(3)` produces 0, 1 and 2, one per time round the loop, and `i` takes each value in turn. There is no setup line and no `i += 1` to forget.

In C the same loop is `for (int i = 0; i < 3; i++)`: a start, a condition and an update, all written by hand.

Scratch's **forever** block is `while True`. The condition never becomes false:

```
# Infinite loop

while True:
    print("meow")
```

**Watch out** this loop never ends, and your terminal fills with meows. Press **Control-C** to stop a running program.

Ask the user how many times to meow:

```
meow.py

# Prompts user for n

n = int(input("What's n? "))

for i in range(n):
    print("meow")
```

What if the user types a negative number? We could ask again once:

meow.py

```
# Prompts user again if need be (poor design)

n = int(input("What's n? "))
if n < 0:
    n = int(input("What's n? "))

for i in range(n):
    print("meow")
```

But a stubborn user can type a negative number twice. Use a loop that keeps asking, with `continue` (go back to the top of the loop) and `break` (leave the loop):

meow.py

```
# Uses a loop with continue and break

while True:
    n = int(input("What's n? "))
    if n < 0:
        continue
    else:
        break

for i in range(n):
    print("meow")
```

The `continue` branch is not needed: when the number is good, break; otherwise the loop goes round anyway.

meow.py

```
# Uses a loop with just break

while True:
    n = int(input("What's n? "))
    if n >= 0:
        break

for i in range(n):
    print("meow")
```

In C this pattern is usually a `do ... while` loop, which always runs at least once. Python has no `do ... while`; `while True` with `break` does the same job.

## 16 Functions

Scratch lets you make your own blocks. In Python, `def` defines your own function:

```
def meow():
    print("meow")
```

Then use it like any other function:

```
meow.py

# Abstraction

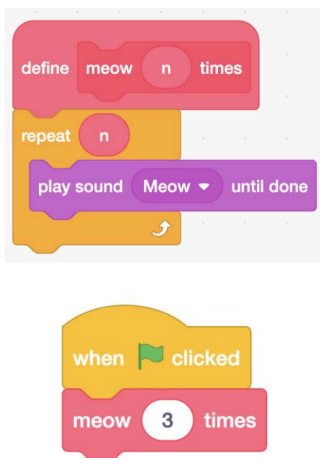
def main():
    for i in range(3):
        meow()

# Meow once
def meow():
    print("meow")

main()
```

Python runs a file from top to bottom. The two `def` blocks only define functions; they run nothing. The last line, `main()`, starts the program. By then both functions exist, so `main` can call `meow` even though `meow` is defined further down. Putting `main` first lets a reader see the big picture straight away.

Hiding the details of meowing behind a name is called **abstraction**. Give the function a **parameter** so it can meow any number of times:



```
meow.py

# Abstraction with parameterization

def main():
    meow(3)

# Meow some number of times
def meow(n):
    for i in range(n):
        print("meow")

main()
```

In C you must state what a function takes and returns, as in `void meow(int n)`, and repeat that line (a **prototype**) at the top of the file if the function comes after `main`. Python needs neither.

A variable's **scope** is the part of the program where it exists. Consider:

meow.py

```
# Demonstrates scope

def main():
    n = 3
    meow(n)

# Meow some number of times
def meow(n):
    for i in range(n):
        print("meow")

main()
```

The `n` made in `main` exists only inside `main`. `meow` can use the number only because `main` passes it as an argument. Inside `meow`, the parameter `n` is a separate name that happens to hold the same value. If `meow` took no parameter and used `n` anyway, Python would stop with `NameError: name 'n' is not defined`.

A function can also **return** a value to whoever called it, as `input` does. Here a function keeps asking until the number is positive, then hands it back:

meow.py

```
# Return value

def main():
    n = get_positive_int()
    meow(n)

# Get number of meows
def get_positive_int():
    while True:
        n = int(input("Number: "))
        if n >= 1:
            return n

# Meow some number of times
def meow(n):
    for i in range(n):
        print("meow")

main()
```

`return n` both leaves the loop and sends `n` back to `main`, so no `break` is needed.

## 17 Correctness, design, style

Judge code on three axes:

| Axis        | Question                         | How to check                                                                                       |
|-------------|----------------------------------|----------------------------------------------------------------------------------------------------|
| Correctness | Does the code do what it should? | Run it with normal input and with strange input: 0, negative numbers, capital letters.             |
| Design      | Is it well designed?             | Look for repeated lines and questions you do not need to ask. Could it be shorter and clearer?     |
| Style       | Is it neat and consistent?       | Clear names, 4-space indents, spaces around <code>=</code> . Python's style guide is called PEP 8. |

In CS50 these are checked by the tools `check50` , `design50` and `style50` .

## 18 Mario

How do you approach a new problem? Start small, get it working, then improve it. Imagine you want to draw parts of the world of Super Mario Bros. How could you print these four question blocks in a row?



mario.py

```
# Prints a row of 4 question marks

print("????")
```

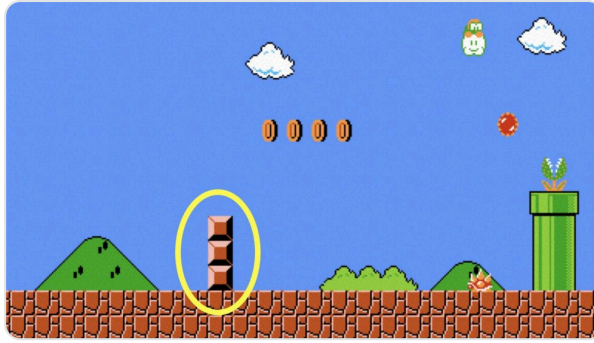
With a loop, the number of blocks becomes something you can change. Pass `end=""` so the question marks stay on one line, then print a single new line at the end:

mario.py

```
# Prints a row of 4 question marks with a loop

for i in range(4):
    print("?", end="")
print()
```

The same idea gives a column of three bricks. This time the automatic new line is what we want:

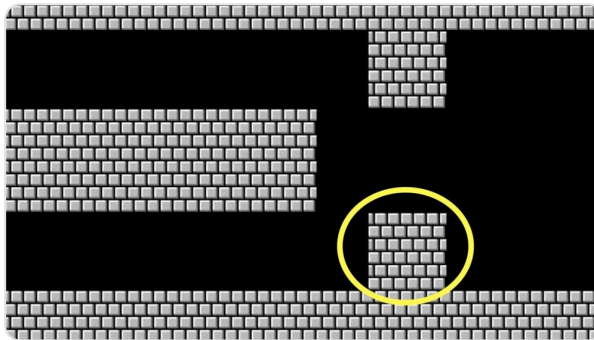


mario.py

# Prints a column of 3 bricks with a loop

```
for i in range(3):
    print("#")
```

Combine the two ideas to make a three-by-three grid of bricks:



mario.py

# Prints a 3-by-3 grid of bricks with nested loops

```
for i in range(3):
    for j in range(3):
        print("#", end=" ")
    print()
```

One loop is inside the other. The outer loop, with `i`, chooses the row. For each row, the inner loop, with `j`, prints three bricks. After each row, `print()` moves to a new line.

What if the size should never change? Python has no way to lock a variable. By convention, a name in capitals means "do not change this", and it is called a **constant**:

mario.py

# Prints a 3-by-3 grid of bricks with nested loops using a constant

`N = 3`

```
for i in range(N):
    for j in range(N):
        print("#", end=" ")
    print()
```

In C `const int n = 3;` makes a true constant; the compiler refuses any change to it.

As before, we can move part of the work into a function of its own:

```
mario.py

# Helper function

def main():
    n = 3
    for i in range(n):
        print_row(n)

def print_row(width):
    print("#" * width)

main()
```

Multiplying a string repeats it, so `"#" * 3` is `###`. That replaces the whole inner loop.

## 19 Operators

Python supports these arithmetic operators:

| Operator        | Meaning                                  | Example                                   |
|-----------------|------------------------------------------|-------------------------------------------|
| <code>+</code>  | addition                                 | <code>7 + 2</code> is <code>9</code>      |
| <code>-</code>  | subtraction                              | <code>7 - 2</code> is <code>5</code>      |
| <code>*</code>  | multiplication                           | <code>7 * 2</code> is <code>14</code>     |
| <code>/</code>  | division                                 | <code>7 / 2</code> is <code>3.5</code>    |
| <code>//</code> | division, rounded down to a whole number | <code>7 // 2</code> is <code>3</code>     |
| <code>%</code>  | remainder                                | <code>7 % 2</code> is <code>1</code>      |
| <code>**</code> | power                                    | <code>2 ** 10</code> is <code>1024</code> |

Make a calculator. Type `code calculator.py` :

calculator.py

```
# Addition with int

# Prompt user for x
x = int(input("What's x? "))

# Prompt user for y
y = int(input("What's y? "))

# Add numbers
z = x + y

# Print the result
print(z)
```

**Watch out** without `int`, typing 1 and 2 prints `12`. On strings, `+` joins them. The type of a value decides what an operator does.

The variable `z` is used only once, so we can do the addition inside `print`:

calculator.py

```
# Addition with int, without third variable

x = int(input("What's x? "))
y = int(input("What's y? "))

print(x + y)
```

Or double a number:

calculator.py

```
# Doubles a number

x = int(input("What's x? "))

print(x * 2)
```

## Overflow

In most languages, an integer is stored in a fixed number of bits, so it can only count so high. This program keeps doubling some money:

dollars.py

```
# Doubling forever

dollars = 1
while True:
    s = input(f"Here's ${dollars:,.}. Double it and give to next person? ")
    if s == "y":
        dollars *= 2
    else:
        break
print(f"Here's ${dollars:,.}")
```

In C, an `int` stops at 2,147,483,647. One more doubling and it **overflows**: the bits wrap around and the program shows a negative number or zero. Switching to a `long` only pushes the limit further away.

Python's `int` has no fixed size. It grows as needed, so this program never overflows; `2 ** 100` is `1267650600228229401496703205376`. Overflow still matters, because most other languages and much of the hardware under them have it. It has caused real failures:

- **Boeing 787**: a counter in the generator control units overflowed after 248 days of continuous power, which could shut off the plane's electrical power. The fix was to restart the system regularly.
- **Pac-Man**: the level number is stored in 8 bits (0 to 255). On level 256 it overflows and half the screen fills with garbage.
- **Year 2000**: years were stored as two digits, so after 99 came 00, and the year 2000 could be read as 1900.
- **Year 2038**: many systems count time as seconds since 1 January 1970 in a signed 32-bit integer. That runs out at 03:14:07 UTC on 19 January 2038 and wraps to 13 December 1901.

## Truncation

An `int` cannot hold a fraction. Dividing with `//` throws away everything after the decimal point, which is called **truncation**:

calculator.py

```
# Division, with and without truncation

x = int(input("What's x? "))
y = int(input("What's y? "))

print(x / y)      # 7 and 2 give 3.5
print(x // y)     # 7 and 2 give 3
```

You can also convert between types. `float(7)` is `7.0`, and `int(3.9)` is `3`: converting to `int` truncates too.

In C `/` on two ints truncates without telling you, so `7 / 2` is `3`. You must **cast** one of them, as in `(float) x / y`, to get `3.5`. In Python, `/` always gives a float, and you ask for truncation on purpose with `//`.

## Floating-point imprecision

A `float` also has a fixed number of bits, but there are infinitely many numbers between 0 and 1. So most decimals are stored as the nearest value a float can hold. Show 50 decimal places of  $1 \div 3$ :

```
calculator.py
```

```
# Floats

x = float(input("What's x? "))
y = float(input("What's y? "))

print(f"{x / y:.50f}")
```

```
$ python calculator.py
What's x? 1
What's y? 3
0.33333333333333331482961625624739099293947219848633
```

The 3s stop after about 16 places. Even `0.1 + 0.2` gives `0.30000000000000004`.

**Watch out** never test whether two floats are equal with `==`. For money, count in whole paise (or cents) using an `int`.

Pay attention to the types you use. They decide which values your program can hold, and choosing the wrong one has caused real disasters.

---

## 20 Summing up

In this lesson you took the building blocks from Scratch and used them in Python. You learned:

- how to write and run your first Python program;
- how to use the command line;
- about functions that come built into Python, such as `print` and `input`;
- how to use variables, conditionals and loops;
- how to write your own functions to simplify and improve your code;
- how to judge code on three axes: correctness, design and style;
- how to add comments to your code;
- how types and operators work, and what can go wrong: overflow, truncation and floating-point imprecision.

See you next time!

---

Adapted from [CS50x Lecture 1 notes](#) by David J. Malan and the CS50 team, Harvard University, licensed under [CC BY-NC-SA 4.0](#). Rewritten in Python for cs17.org. Scratch, flowchart and Mario images are from the CS50 lecture slides.