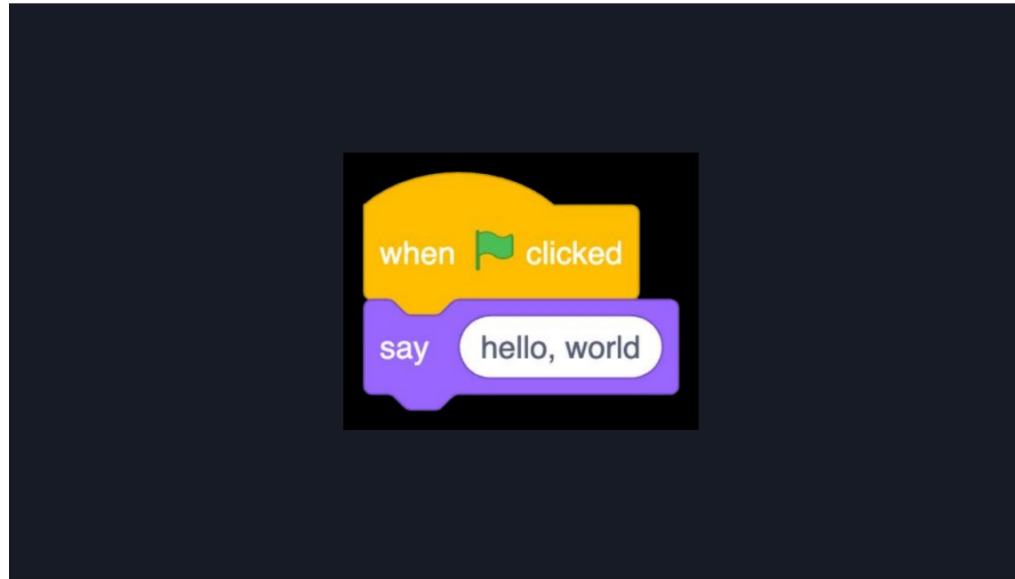


CS17.ORG • LECTURE 1

Python

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

Welcome. Last term you built games in Scratch by snapping blocks. Today the same ideas come back as text: functions, conditionals, loops, variables. We start with Python. Everything you learned in Scratch still applies; only the way we write it changes.



This is where you already are. Two blocks: an event (green flag) and an action (say). Ask: what does this program do? Every program today does the same kind of thing, but typed instead of dragged.

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

hello.py

The same program in Python. One line. "print" plays the role of "say". The text in quotes is a string. The file is called hello.py: .py tells everyone it is Python.

source code

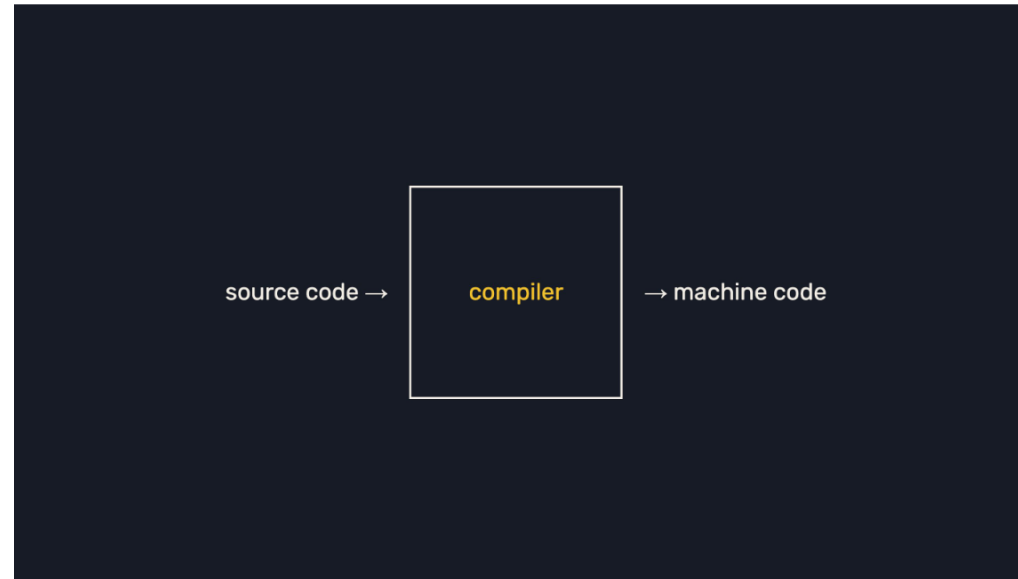
what people write and read

Source code is the text we write. Humans can read it. The computer cannot run it directly.

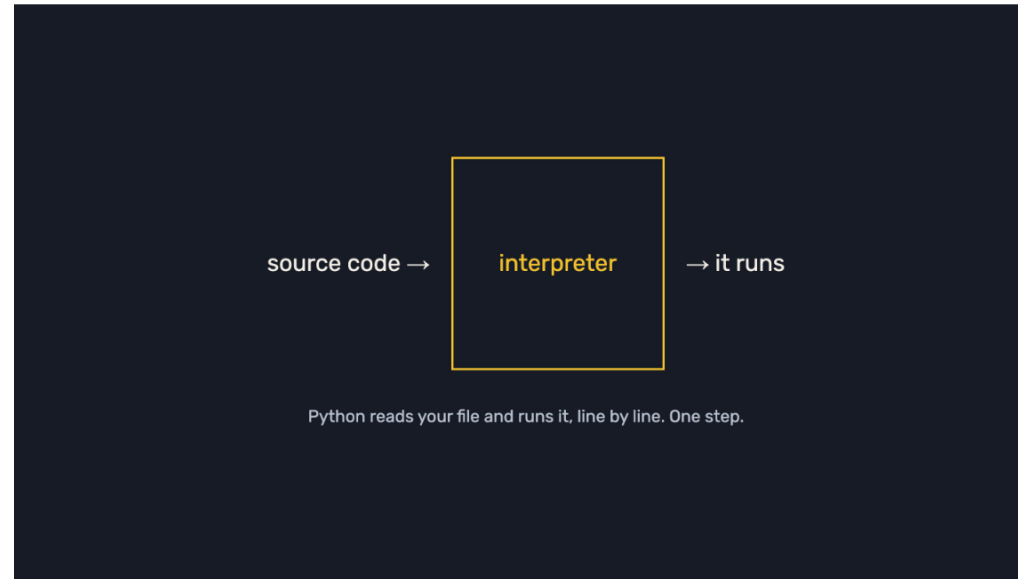
```
01111111 01000101 01001100 01000110 00000010 00000001
00000000 00000000 00000010 00000000 00111110 00000000
10110000 00000101 01000000 00000000 00000000 00000000
11010000 00010011 00000000 00000000 01000000 00111000
00001001 00000000 01000000 00000000 00100100 00100001
...
```

machine code

Machine code: the 0s and 1s the CPU actually runs. Recall binary from the start of the course. Nobody writes this by hand any more. So we need a translator.



The CS50 picture: input, a black box, output. In C the box is a compiler. It turns the whole file into machine code once; then you run that file. Two steps: make hello, then ./hello.



Python's box is an interpreter. It reads `hello.py` and runs it right away, line by line. No separate machine-code file to manage. Slower for the computer, faster for us. For learning, fast feedback wins.

OUR TOOLS

An editor and a terminal

GUI

Graphical user interface. Windows, menus, the mouse. Where we **write** code.

CLI

Command-line interface. You type commands. Where we **run** code.

Recall Worksheet 21: ls, cd, mkdir, cp, mv, rm. The terminal is a CLI. The editor is a GUI. We use both together: write in the editor, run in the terminal.

IN THE TERMINAL

```
$ code hello.py  
$ python hello.py  
hello, world
```

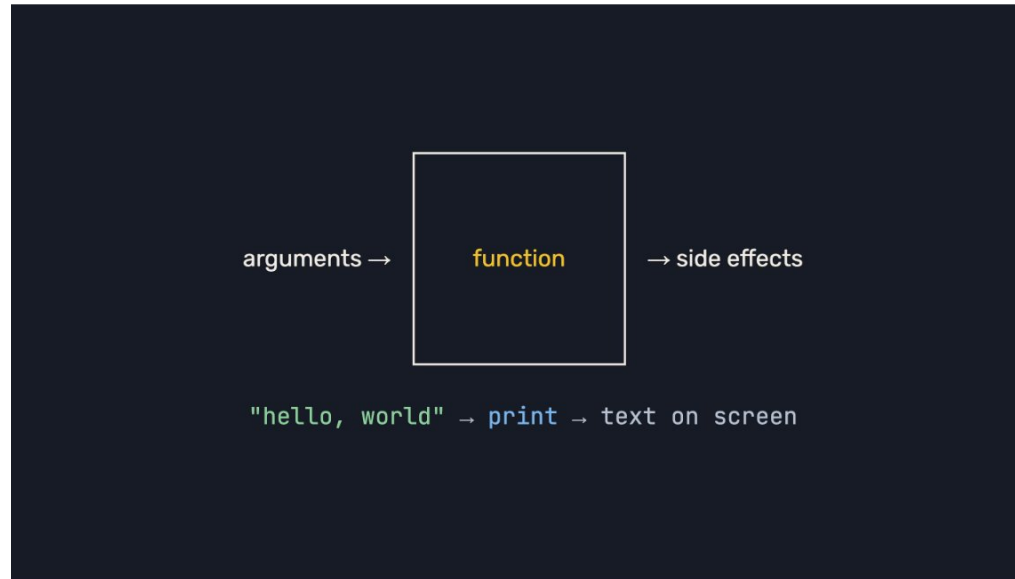
C needs two steps: `make hello` then `./hello`. Python needs one.

Live demo. Create `hello.py`, type the line, save, run. Then break it: delete the closing quote and run again. Read the `SyntaxError` together. Then try `Print` with a capital P: `NameError`. Errors are normal; reading them is a skill. On some Macs the command is `python3`.

PART 1

Functions

First idea: functions. A function is a named action. print is one. Scratch's say block is one.



Inputs to a function are called arguments. Some functions do something you can see: a side effect. `print`'s side effect is text appearing on the screen. It hands nothing back to your program.

READING ONE LINE

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

print

the function's name

()

"call it now", with these arguments

"hello, world"

a string: text in quotes

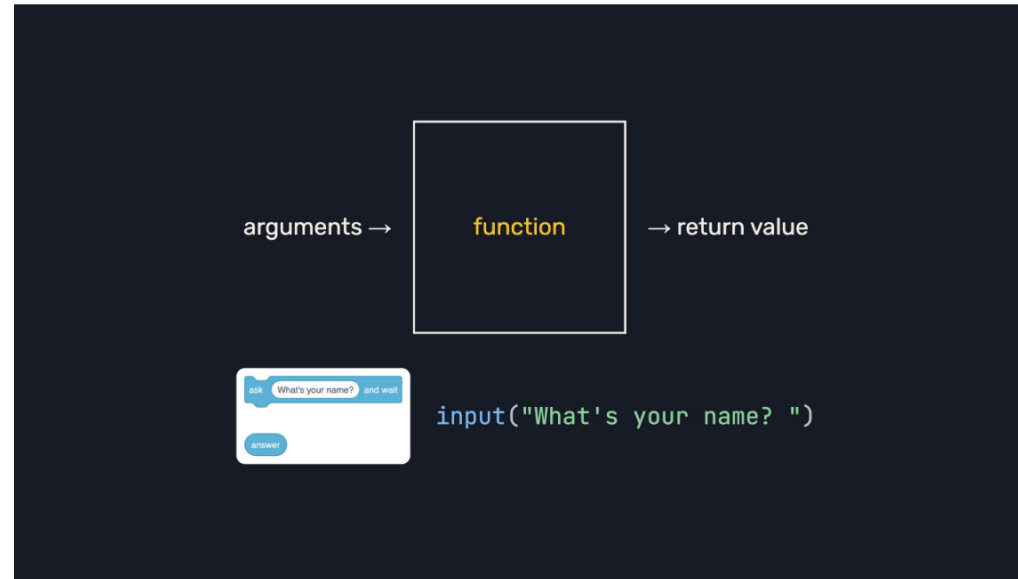
Name the parts. `print` is the function. Parentheses mean call it. The string in quotes is the argument. Single or double quotes both work in Python; pick one and stay consistent.

ESCAPE SEQUENCES

A backslash means "special"

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

`print` adds a new line automatically, unlike C's `printf`. To stop it, pass `end=""`. We need that trick for Mario later. Demo: `print("hello\nworld")` prints two lines.



Other functions hand a value back: a return value. `input` shows a prompt, waits for the user, and returns what they typed as a string. In Scratch this is the ask block and its answer reporter. CS50 needs `get_string` from its own library; Python has `input` built in.

KEEP THE ANSWER IN A VARIABLE



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

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

The Scratch version on the left: ask, then join "hello, " with answer. Python on the right. The = sign is assignment, not equals: copy the value on the right into the name on the left. Read it right to left. + is Python's join block. No type needed: C writes string answer; Python just writes answer. Common bug: forgetting the space after "hello," gives "hello,David".

THREE WAYS TO SAY THE SAME THING

```
print("hello, " + answer)
print("hello,", answer)
print(f"hello, {answer}")
```

The `f` makes a format string: `{}` holds a value to fill in.

Join with `+`. Or pass two arguments: `print` puts a space between them. Or use an f-string: the curly braces are filled in. f-strings play the role of C's `%s` format codes. We use f-strings from now on.

VARIABLES

set counter to 0
change counter by 1

```
counter = 0  
counter = counter + 1  
counter += 1
```

No `counter++` in Python. Use `+= 1` and `-= 1`.

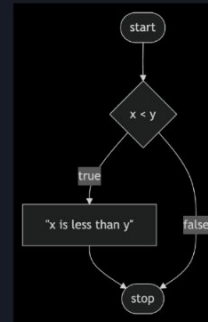
Same orange variable blocks from Scratch. `counter = counter + 1` looks wrong as maths, but read it as: take counter, add 1, store it back in counter. `+=` is the short form. C has `counter++`; Python does not.

PART 2

Conditionals

Second idea: making decisions. Scratch's if and if-else blocks.

IF

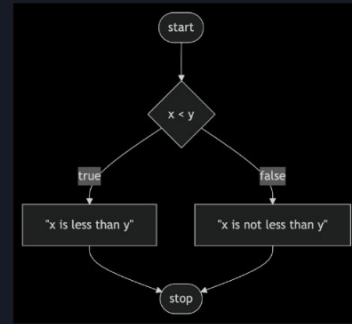


```
if x < y:  
    print("x is less than y")
```

The colon opens the block. The **indent** is the block, like the C-shape in Scratch.

Trace the flowchart: one question, two paths. Compare with Scratch's C-shaped if block. In C you write parentheses and curly braces. In Python: no parentheses needed, a colon, then indent four spaces. The indentation is not style; it is the syntax. Wrong indent = IndentationError.

IF / ELSE



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

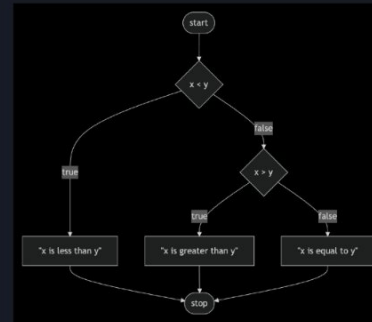
A fork in the road: exactly one of the two branches runs. else also ends in a colon.

IF / ELIF / ELIF

```
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")
```

elif is Python's "else if". Note == for comparison: one = assigns, two == compares. Ask the room: is the last question necessary? If x is not less and not greater, what else can it be?

BETTER DESIGN: ASK ONLY WHAT YOU MUST



```
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")
```

Same output, one less question. The flowchart shows it: at most two diamonds. Both versions are correct; this one is better designed. Correct is not the only goal.

OPERATORS

Compare and combine

Write	Means
=	assign
==	equal to?
!=	not equal to?
< <=	less (or equal)?
> >=	greater (or equal)?

Python	C
and	&&
or	
not	!
True	true
False	false

The comparison operators are the same as C. The boolean ones are English words in Python: and, or, not. True and False start with a capital letter. Scratch has the same green and/or/not blocks.

COMPARE.PY

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

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")
```

Live demo. First write it without `int()`. Type 2 and 10: it says x is greater! Because input returns strings, and "2" comes after "10" alphabetically. `int()` converts the string to a number. This is the same job as CS50's `get_int`. Then try typing "cat": `ValueError`. We fix that in Mario.

AGREE.PY

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

```
if s == "y" or s == "yes":  
    print("Agreed.")
```

```
elif s in ["n", "no"]:  
    print("Not agreed.")
```

Strings know tricks: `.lower()` makes "YES" into "yes".

CS50 uses a single char here. Python has no separate char type: a character is a string of length one. Two ways to check several answers: `or`, `and` in with a list. `.lower()` is a method: a function that belongs to the string. Ask: what if the user types "Y"? That's why we lower it first.

PART 3

Loops

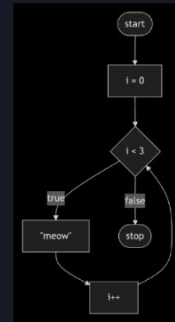
Third idea: doing something again and again. Scratch's repeat and forever blocks. Running example: a cat that meows three times.

WHILE: COUNT DOWN

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

A while loop repeats while its condition is true. Trace it on the board: i is 3, meow, 2, meow, 1, meow, 0, stop. Three meows. What happens if we forget `i -= 1`? It never stops.

WHILE: COUNT UP FROM 0



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

Programmers count from 0. You'll see why with lists.

The flowchart shows the loop as an arrow going back up. Start at 0, go while less than 3. Values of i: 0, 1, 2. Still three meows. Counting from zero is a convention you'll see everywhere.



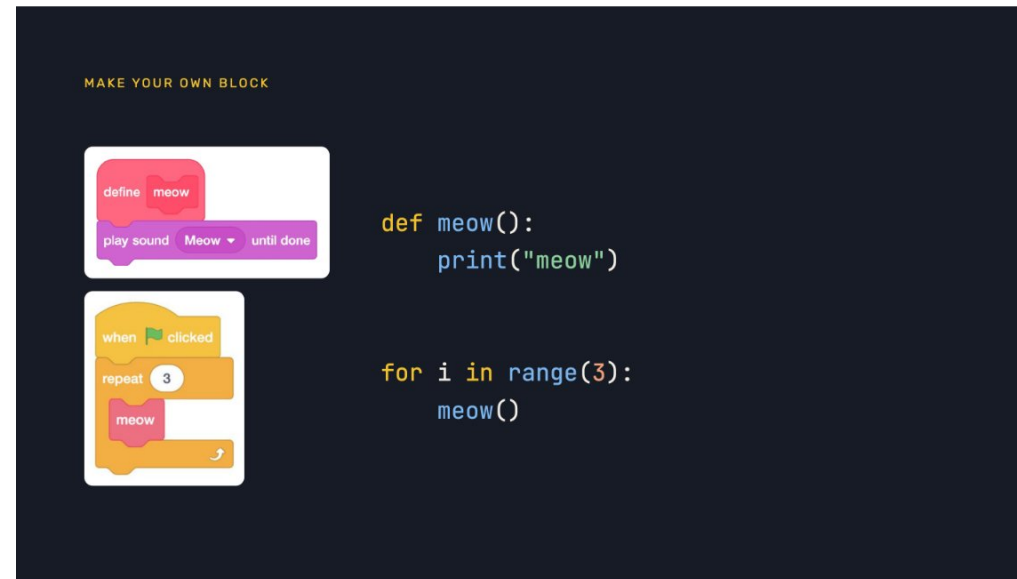
The for loop is the direct match for Scratch's repeat block. `range(3)` produces 0, 1, 2. No setup line and no `i += 1` to forget. Compare with C: `for (int i = 0; i < 3; i++)`. Use for when you know how many times; use while when you don't.

FOREVER

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

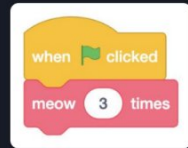
Stuck? Press **Control-C**.

Scratch's forever block. The condition True never becomes false. Run it live, let it scroll, then press Ctrl-C. Students will write infinite loops by accident; now they know the escape. Later, break gets us out from inside.



Scratch's pink My Blocks. `def` defines a new function. Now the loop reads like English: three times, meow. This is abstraction: we hide how meowing works behind a name. Define before you call, or Python says `NameError`.

GIVE IT AN ARGUMENT



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

Now meow takes an argument *n*, like a My Block with an input. Putting the main code in a main function lets us put it at the top, where a reader looks first. The last line, `main()`, actually starts the program. C calls `main` for you; Python does not.

SCOPE

A variable lives inside its function

```
def main():  
    n = 3  
    meow()  
  
def meow():  
    for i in range(n):  
        print("meow")
```

NameError: name 'n' is not defined

meow cannot see main's **n**. Pass it in as an argument instead.

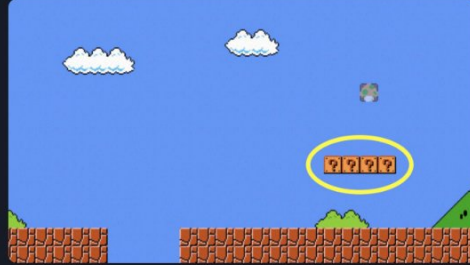
Scope: a variable made inside a function exists only there. Like "for this sprite only" variables in Scratch. The fix is the previous slide: pass **n** as an argument.

PART 4

Mario

Now we combine everything: loops inside loops, and checking what the user types. Our example comes from Super Mario Bros.

A ROW OF QUESTION BLOCKS



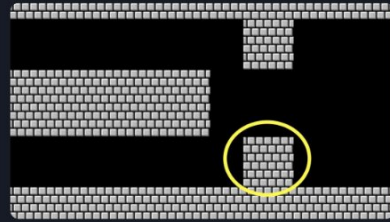
```
for i in range(4):  
    print("?", end="")  
print()  
????
```

Four blocks side by side. Without `end=""` we'd get four lines. The last `print()` moves to a new line at the end. Python shortcut, show after: `print "?" * 4`. Multiplying a string repeats it.



Vertical is the easy case: print's automatic new line does the work. Shortcut: `print("#\n" * 3, end="")`.

A GRID: A LOOP INSIDE A LOOP



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

```
###  
###  
###
```

Outer loop: rows (i). Inner loop: bricks in that row (j). Like reading a page: left to right, then down. Trace i and j on the board. Then: `print("#" * 3)` inside one loop does the same. Ask: what changes for a 5 by 5 grid? Make the size a variable.

ASK AGAIN UNTIL THE ANSWER IS GOOD

```
while True:
    n = int(input("Size: "))
    if n > 0:
        break

for i in range(n):
    print("#" * n)
```

Users type nonsense: -5, 0. Loop forever, and break out once the answer is good. This replaces C's do-while, which Python does not have. Typing "cat" still crashes with ValueError; try/except fixes that in a later lecture. Optional: wrap the first part in def get_size() that returns n.

GOOD CODE

Three questions to ask

Correctness

Does it work, even for odd input like 0 or -5?

Design

Is it simple? No repeated code, no needless questions.

Style

Is it easy to read? Clear names, 4-space indents.

CS50 grades with `check50`, `design50` and `style50`. For us: test with strange input, ask "can this be shorter and clearer?", and keep names and spacing tidy. Python's style guide is called PEP 8.

PART 5

Types and math

Last idea: kinds of values, and where computer maths goes wrong. Python hides some of this, so we look on purpose.

TYPES

Every value has a type

Python	Example	Kind of value	C
int	42	whole number	int, long
float	3.14	decimal number	float, double
str	"hello"	text	string, char
bool	True	yes or no	bool

Ask Python: `type(42)` → `<class 'int'>`

C makes you write the type on every variable. Python works it out, but the types are still there. Four to know today. Live: `type(42)`, `type("42")`, `type(4.2)`, `type(True)`. Note "42" with quotes is a str, not an int.

Python does maths

Write	Means	Example → result
+ - *	add, subtract, multiply	2 * 3 → 6
/	divide	7 / 2 → 3.5
//	divide, drop the decimal	7 // 2 → 3
%	remainder	7 % 2 → 1
**	power	2 ** 10 → 1024

Same as C except: / always gives a float in Python, // is whole-number division, and ** is power (C has no power operator). % is great for "is it even?":
`n % 2 == 0`.

CALCULATOR.PY: SPOT THE BUG

```
x = input("x: ")
y = input("y: ")
print(x + y)
```

```
x: 1
y: 2
12
```

```
x = int(input("x: "))
y = int(input("y: "))
print(x + y)
```

```
x: 1
y: 2
3
```

Run the left one live. $1 + 2 = 12$! input returns strings, and + on strings joins them. Same bug as compare.py. int() converts. The type of a value decides what + means.

TRUNCATION

```
>>> 1 / 3
0.3333333333333333
>>> 1 // 3
0
```

Truncation chops off everything after the point. In C, `1/3` on two ints gives `0`.

Use the Python REPL: type `python` with no file name and get the `>>>` prompt. CS50's calculator divides two ints in C and gets 0, a classic bug. Python's `/` avoids it; `//` asks for it on purpose. Exit the REPL with `exit()` or Ctrl-D.

FLOATING-POINT IMPRECISION

```
>>> print(f"{1/3:.50f}")  
0.33333333333333331482961625624739099293947219848633
```

```
>>> 0.1 + 0.2  
0.30000000000000004
```

A float has a fixed number of bits, so most decimals are stored as the **nearest** value it can hold.

Infinitely many real numbers, finitely many bits: so most decimals are approximations. Show 50 places of $1/3$: the 3s stop and junk appears. Then $0.1 + 0.2$. Lesson: never test floats with `==`. For money, count whole paise or cents as ints.

INTEGER OVERFLOW

Count with only 3 bits:

```
000 → 001 → 010 → 011  
100 → 101 → 110 → 111  
→ 000 (back to 0!)
```

C's `int` stops at 2,147,483,647. Python's `int` just grows:

```
>>> 2 ** 100  
1267650600228229401496703205376
```

With a fixed number of bits, adding 1 to the biggest number wraps around to zero (or to a negative number in C). C's `int` has 32 bits: max 2,147,483,647. Python's `int` has no fixed size, so it never overflows. Floats still have a fixed size, so they are still imprecise. Other languages, and the hardware under Python, still overflow.

OVERFLOW IN REAL LIFE



Boeing 787: a counter overflowed after 248 days powered on.



Pac-Man: level 256.

Boeing 787: a software counter in the generator control units overflowed after 248 days of continuous power, which could shut off electrical power. The fix was to reboot the plane regularly. Pac-Man stored the level in 8 bits (0 to 255). On level 256 the counter overflows and half the screen turns to garbage.

NOT ENOUGH DIGITS

Two famous date bugs

Year 2000

99 → 00

Years stored as 2 digits. After 1999, computers could read 2000 as **1900**.

Year 2038

2^{31} - 1 seconds

Time as 32-bit seconds since 1970. After **19 January 2038** it wraps to 13 December 1901.

Both are overflow. Y2K: saving two digits of memory per date. Billions were spent fixing it before 1 January 2000. Y2038: many systems count seconds since 1 January 1970 in a signed 32-bit int. The fix is 64 bits. Some of you will help fix this one.

RECAP

Scratch ideas, Python words

Idea	Scratch	Python
Function	say	<code>print("hi")</code>
Return value	ask, answer	<code>input("Name? ")</code>
Variable	set counter to 0	<code>counter = 0</code>
Conditional	if, if-else	<code>if / elif / else</code>
Loop	repeat, forever	<code>for, while True</code>
Own function	My Blocks	<code>def meow(n):</code>

Everything today was a Scratch idea with new spelling. Plus three new warnings: input gives strings, floats are imprecise, fixed-size numbers overflow.

THIS WEEK

Practice

1. hello.py

Ask for a name and greet it with an f-string.

2. mario.py

Ask for a height from 1 to 8. Print a right-aligned pyramid of #.

3. even.py

Ask for a number. Say if it is even or odd. Use %.

4. coins.py

Ask for rupees owed. Print the fewest ₹10, ₹5, ₹2, ₹1 coins. Use // and %.

Problem 2 is the CS50 Mario problem. Hint: row i has $(\text{height} - i)$ spaces then i hashes. Problem 4 adapts CS50's Cash to rupees. Stretch: re-ask on bad input everywhere, using `while True` and `break`.