CMPU 100 · Programming with Data

Expressions, Values, and Names

Class 2

Where are we?

A *program* instructs a computer to do something.

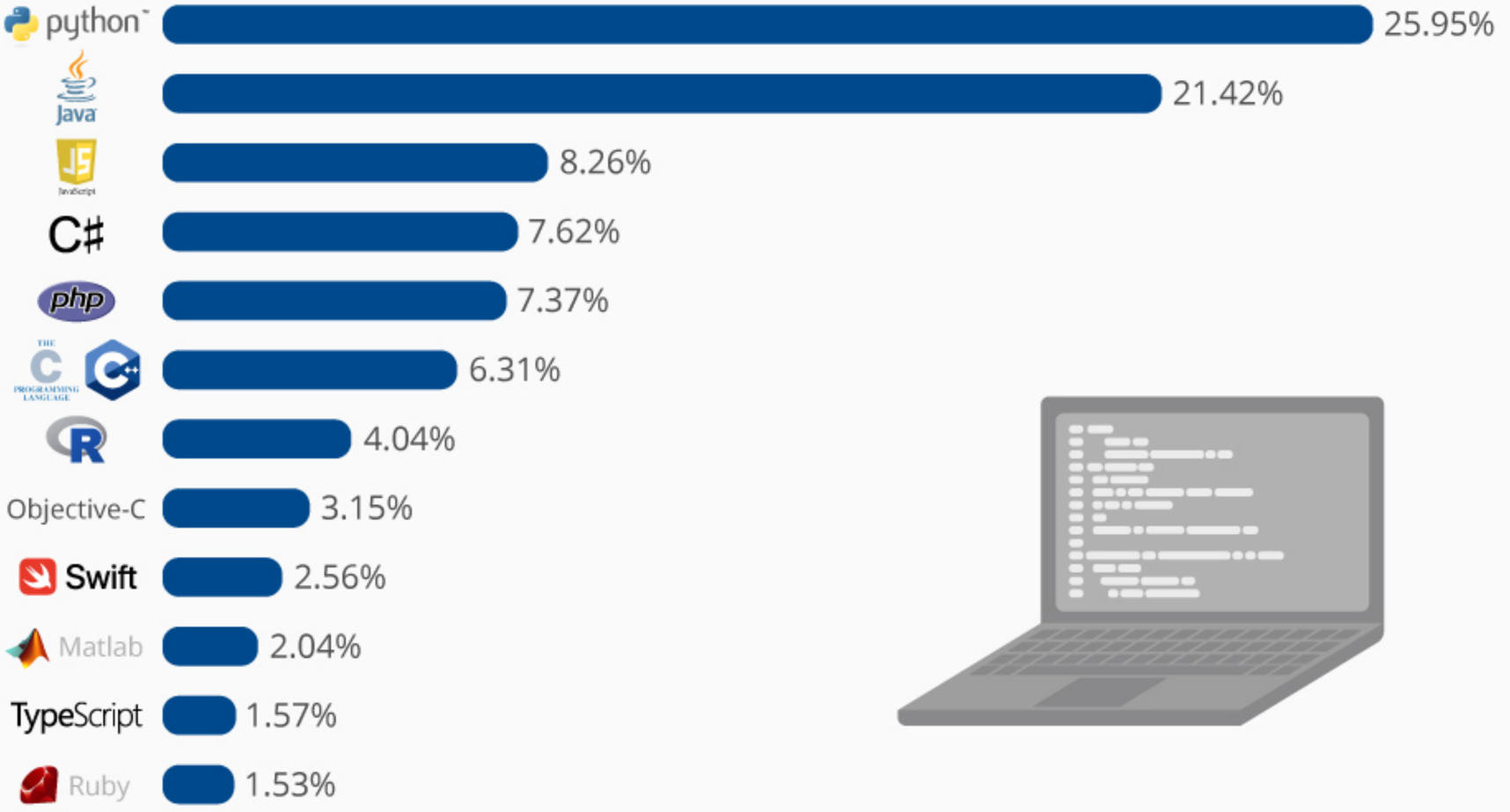
For the computer to carry out these instructions, they need to be *precise*.

But programs also need to be understood by people, so they need to be *readable*!

We write a program in a *programming language* and we run it in a *programming environment*.

The Most Popular Programming Languages

Share of the most popular programming languages in the world*



* Based on the PYPL-Index, an analysis of Google search trends for programming language tutorials.

@StatistaCharts

Source: PYPL

statista

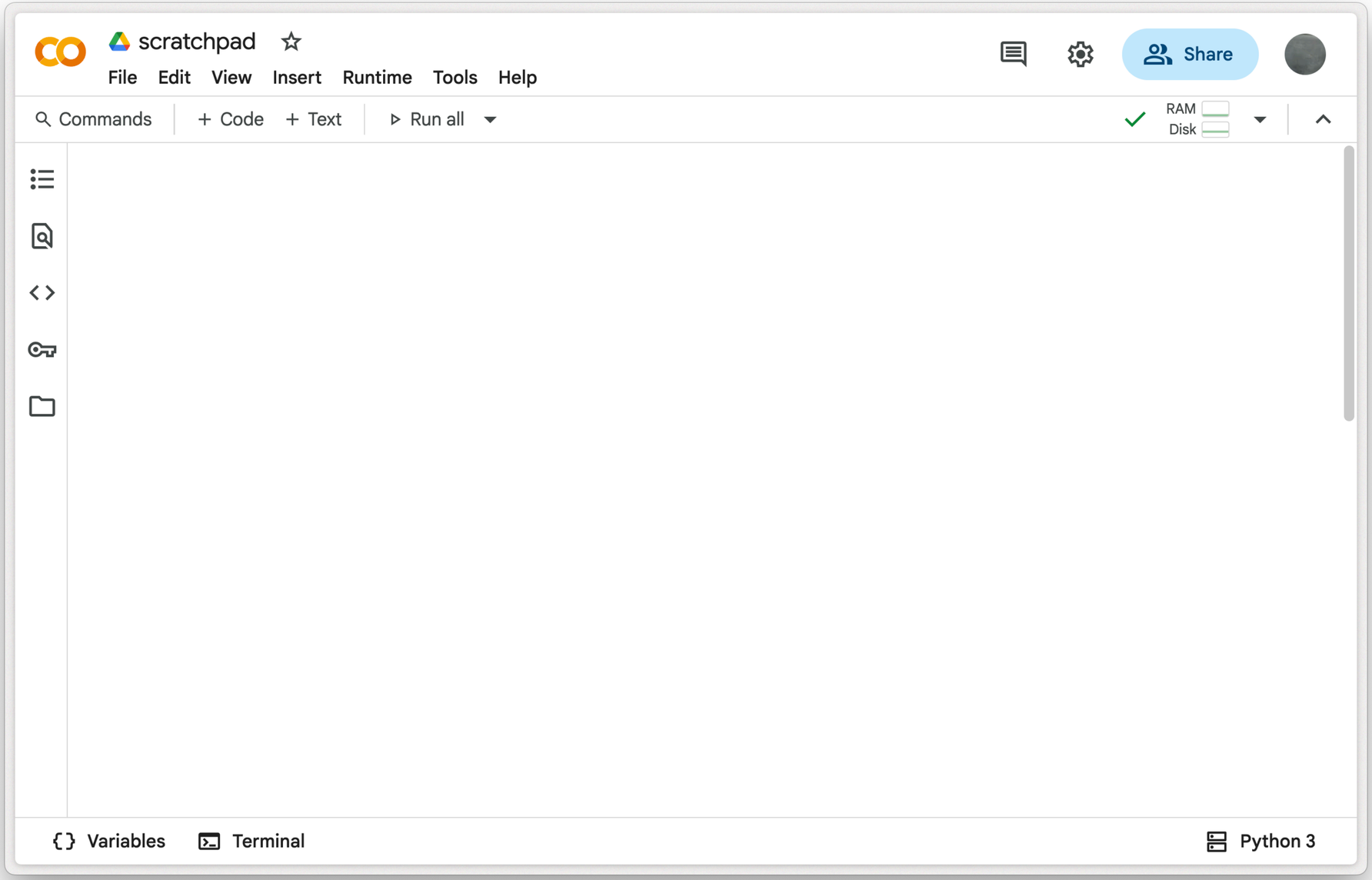
```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

```java
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

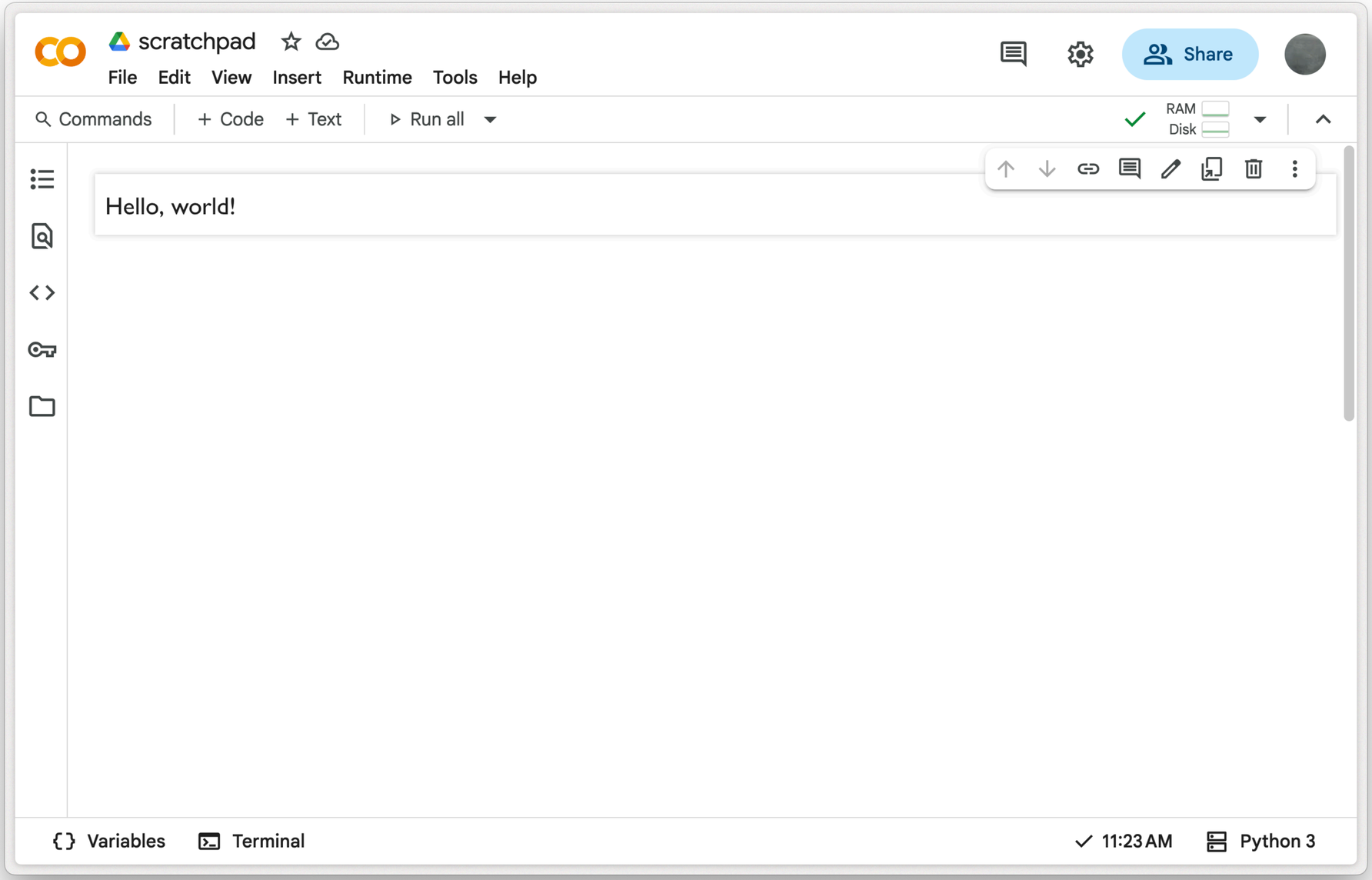
😟

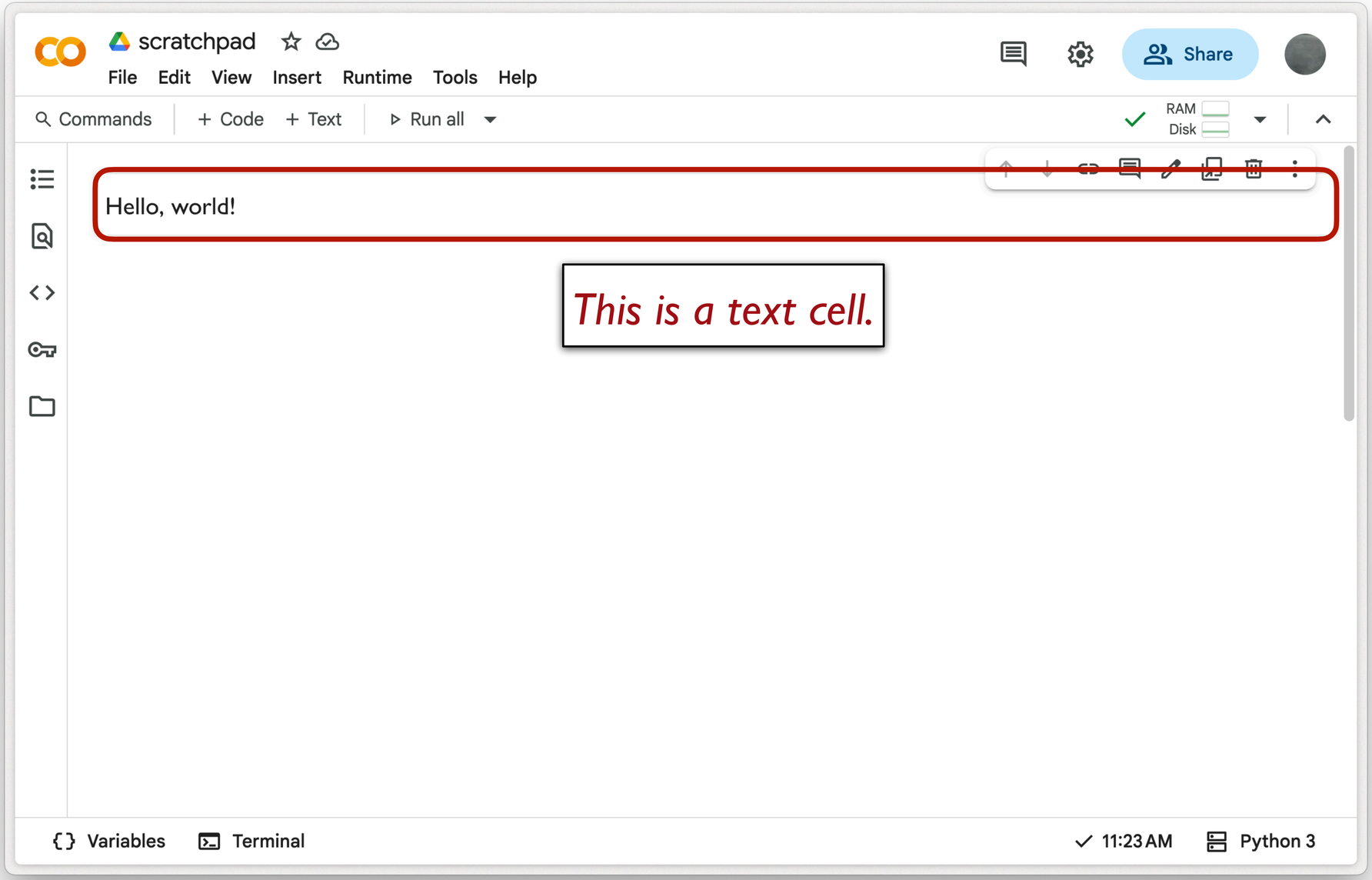
```python
print("Hello, world!")
```

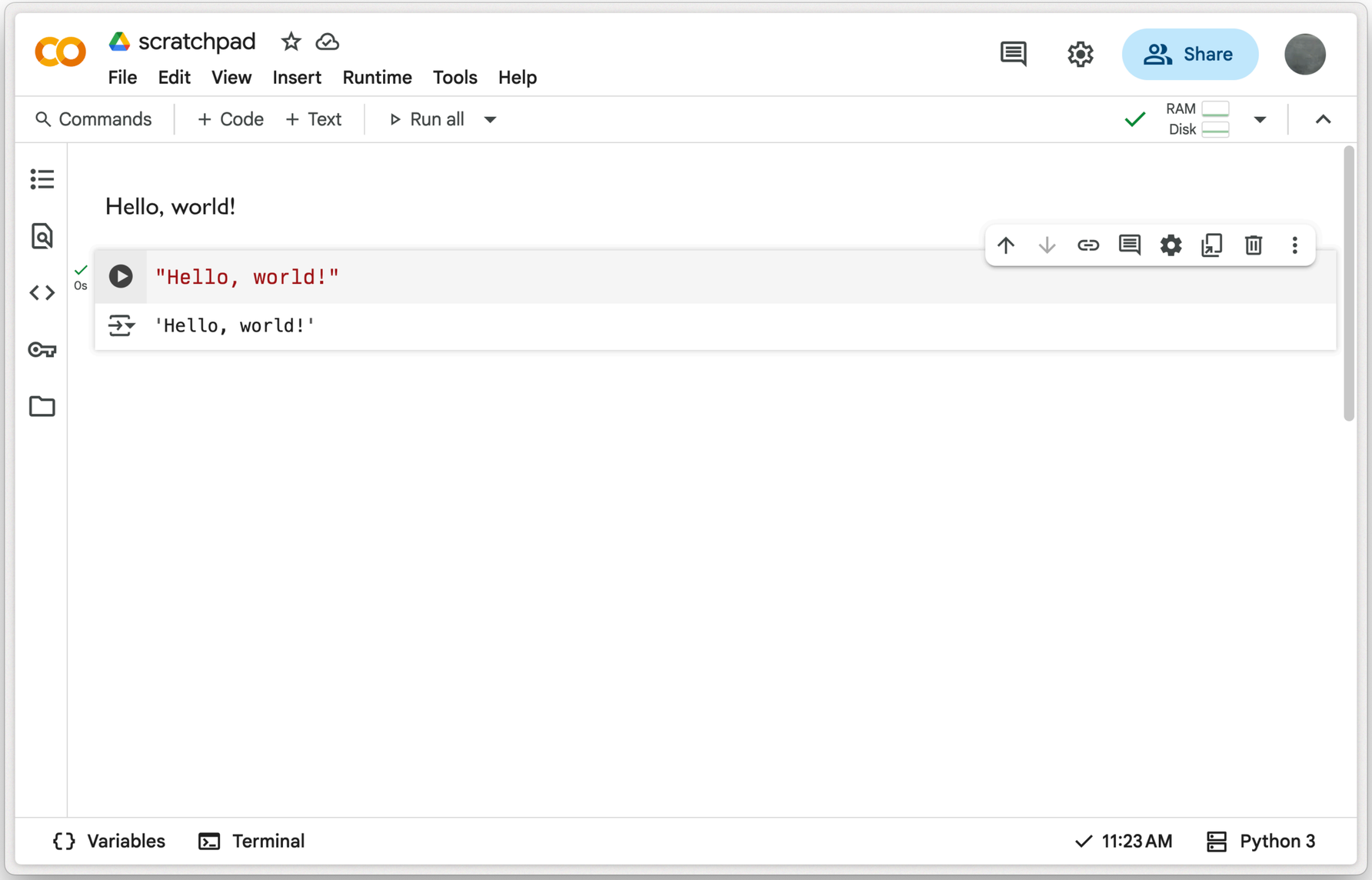
😔

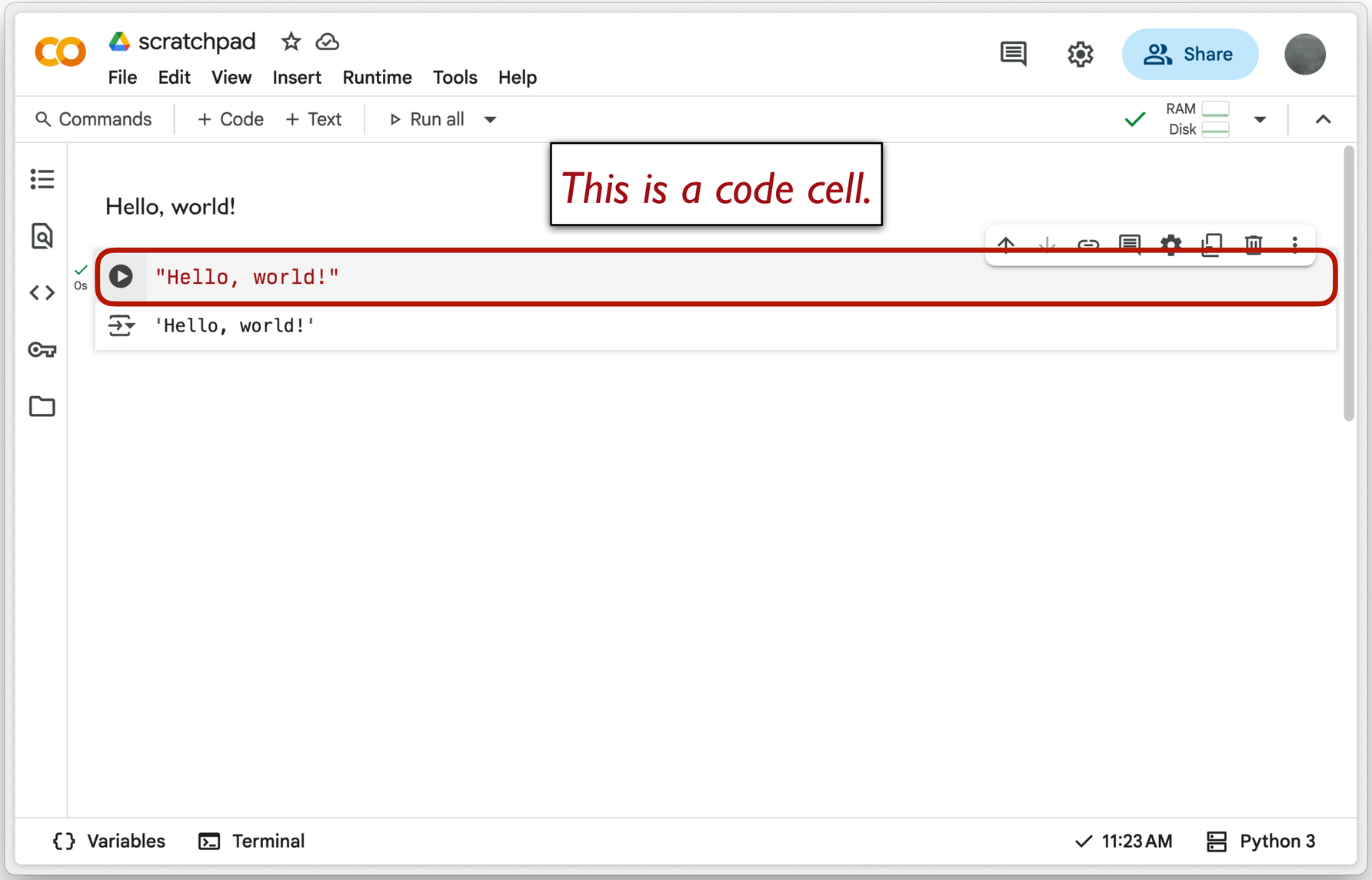


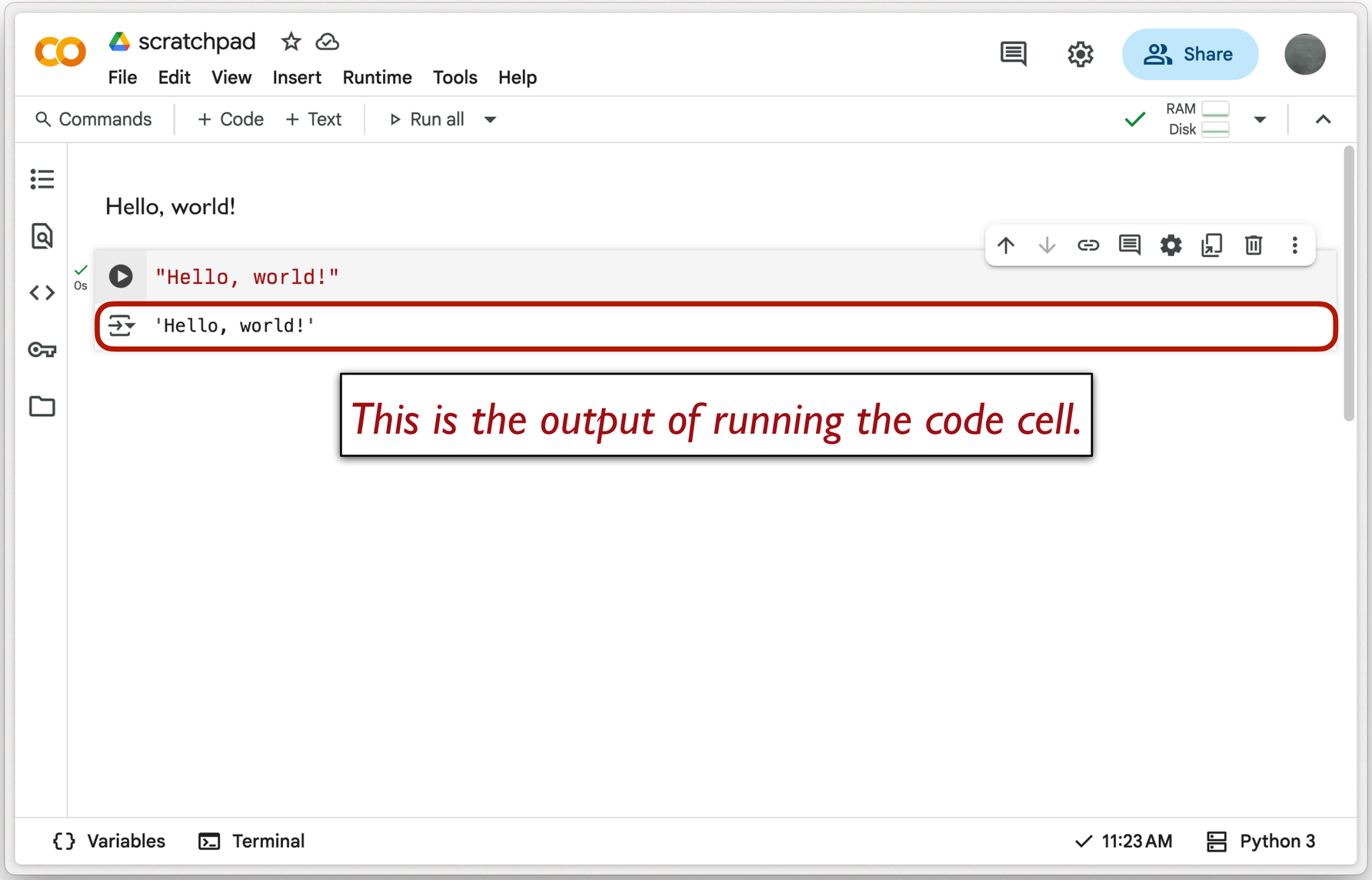
colab.research.google.com


scratchpad
File Edit View Insert Runtime Tools Help
Share
Commands
+ Code
+ Text
Run all
RAM
Disk
Hello, world!
Variables
Terminal
11:23 AM
Python 3


scratchpad
File Edit View Insert Runtime Tools Help
Share
Commands
+ Code
+ Text
Run all
RAM
Disk
Hello, world!
This is a text cell.
Variables
Terminal
11:23 AM
Python 3


scratchpad
File Edit View Insert Runtime Tools Help
Share
Commands
+ Code
+ Text
Run all
RAM
Disk
Hello, world!
"Hello, world!"
0s
'Hello, world!'
Variables
Terminal
11:23 AM
Python 3


scratchpad
File Edit View Insert Runtime Tools Help
Share
Commands
+ Code
+ Text
Run all
RAM
Disk
Hello, world!
This is a code cell.
0s
"Hello, world!"
'Hello, world!'
Variables
Terminal
11:23 AM
Python 3

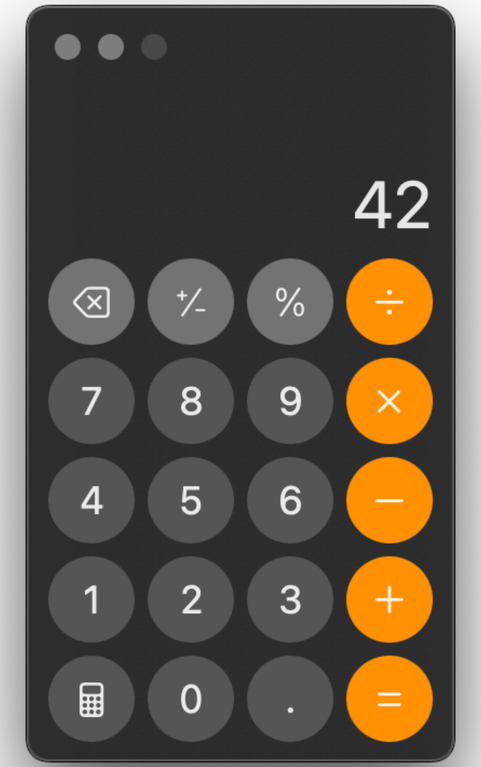

scratchpad
File Edit View Insert Runtime Tools Help
Share
Commands
+ Code
+ Text
Run all
RAM
Disk
Hello, world!
"Hello, world!"
0s
'Hello, world!'
This is the output of running the code cell.
Variables
Terminal
11:23 AM
Python 3

Jupyter notebooks are quite recent – they're the hot format for work in data science – but the idea of interleaving text with code dates back to Donald Knuth's introduction of *literate programming* in 1984.

I guess you guys aren't ready for that yet
but your kids are gonna love it.

Expressions

To start with, you can think of Python like a calculator.



Calculators take *expressions* and compute *values*.

Calculators take *expressions* and compute *values*.

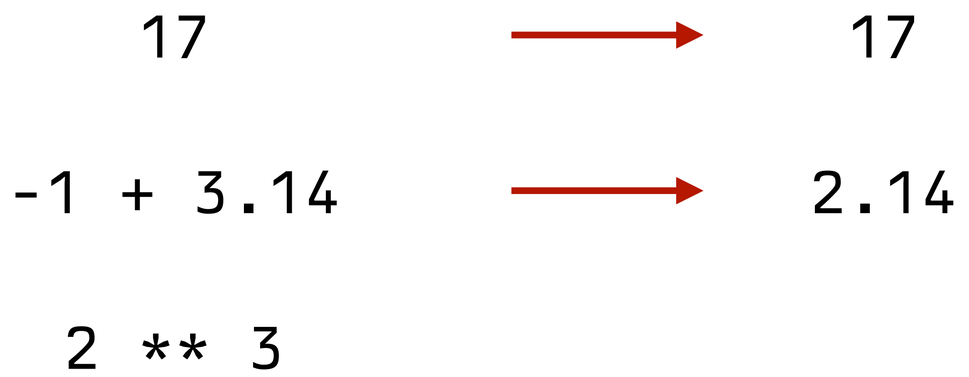
17

Calculators take *expressions* and compute *values*.

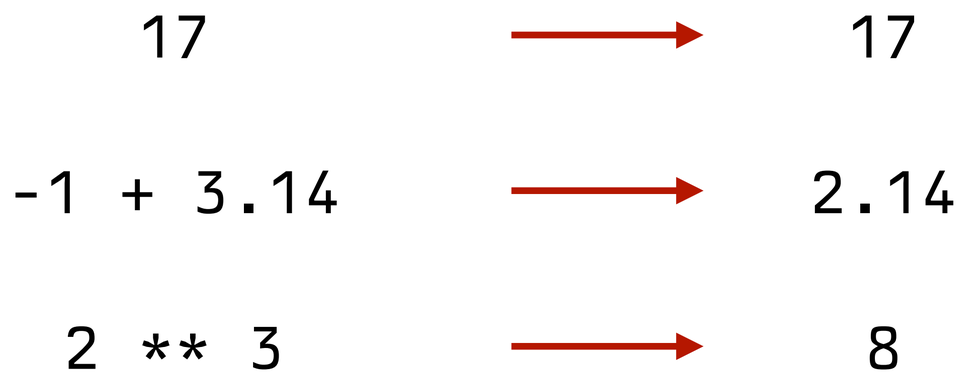
Calculators take *expressions* and compute *values*.

Calculators take *expressions* and compute *values*.

Calculators take *expressions* and compute *values*.



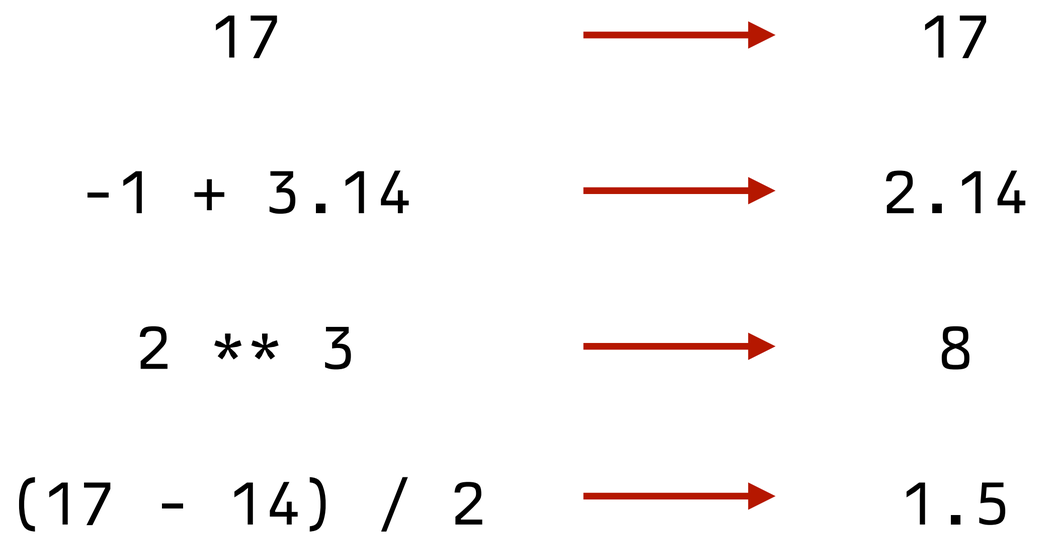
Calculators take *expressions* and compute *values*.



Calculators take *expressions* and compute *values*.

```
17          →  17

-1 + 3.14   →  2.14

2 ** 3      →  8

(17 - 14) / 2
```

Calculators take *expressions* and compute *values*.



`(3 + 4) * (5 + 1)` is an *expression* – a computation that produces an answer.

A program just consists of one or more computations you want to run.

An individual number like **17** is a *value* – it can't be computed any further.

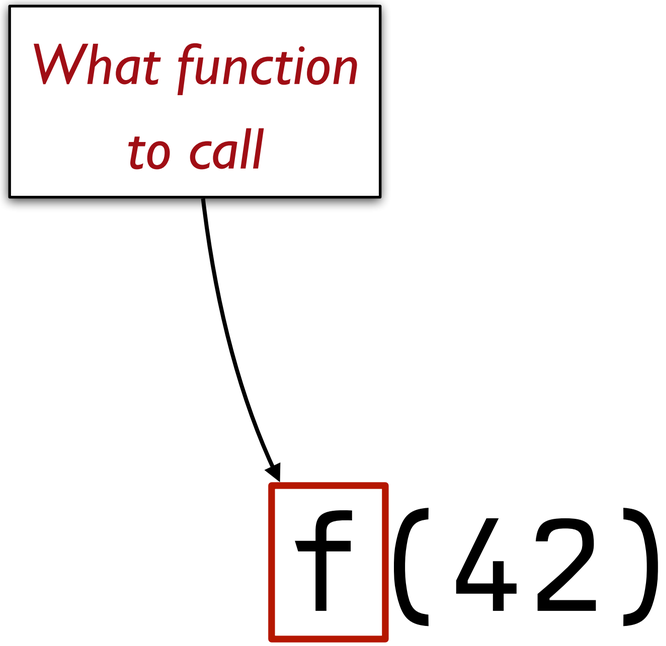
Mathematical expressions in Python use the same *order of operations* that you learned in school (PEMDAS):

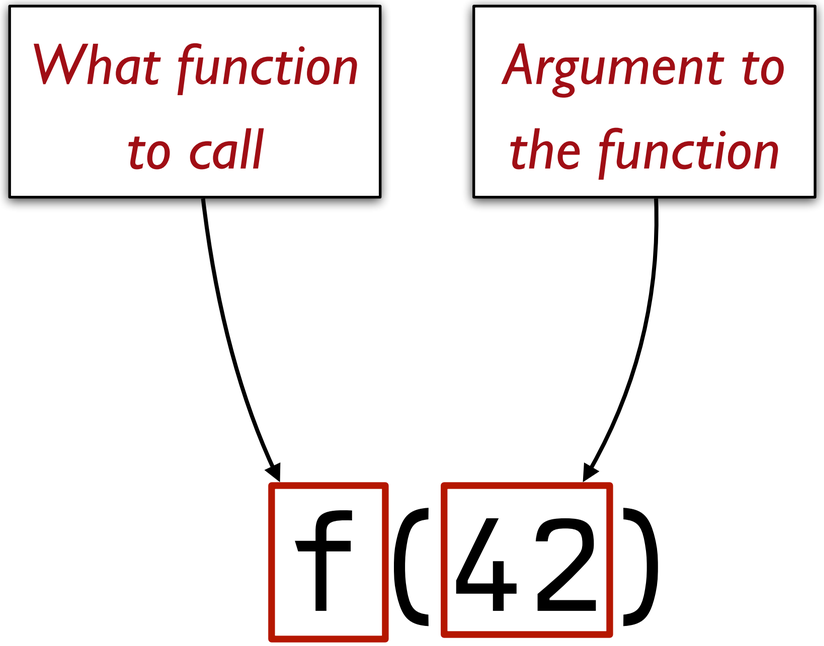
```
60 /  2 * 3   → 90.0
60 / (2 * 3) → 10.0
```

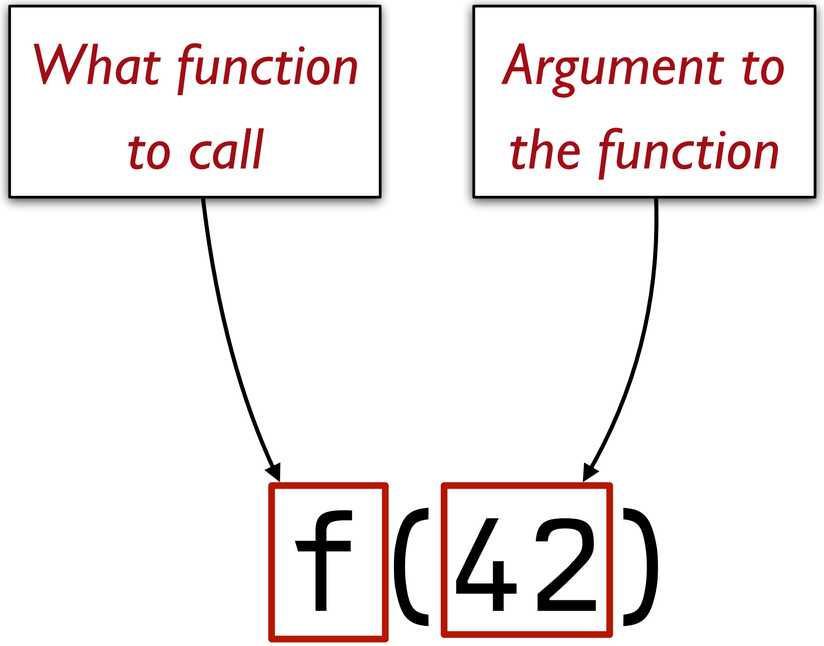
Whenever you're not sure which operator is evaluated first – or you want to fix a certain order – use *parentheses*.

Call expressions

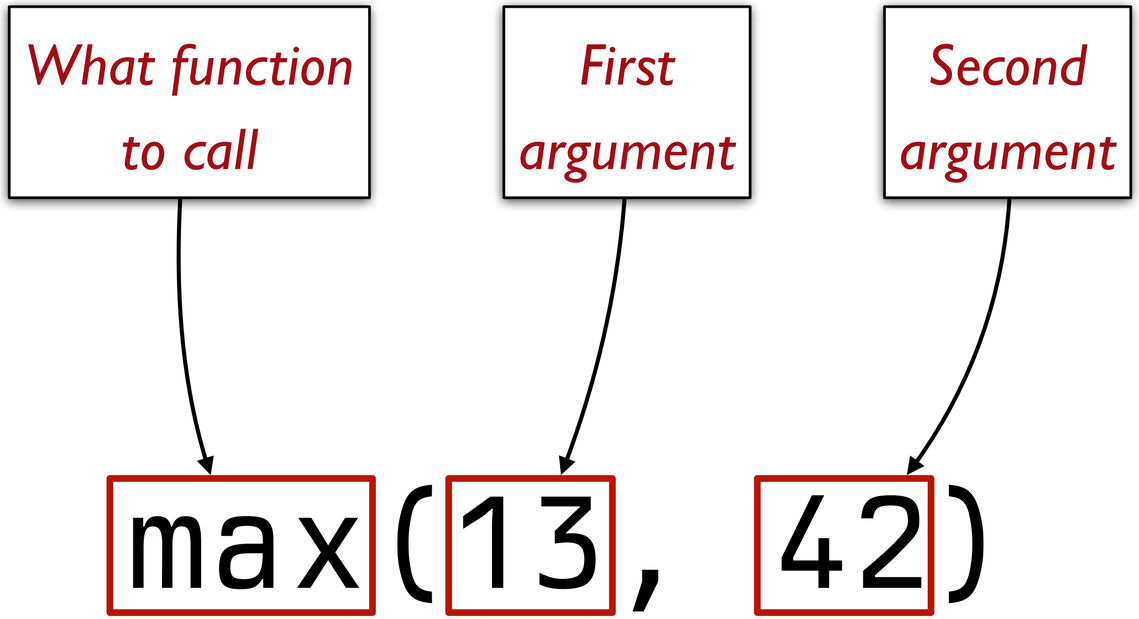
f(42)

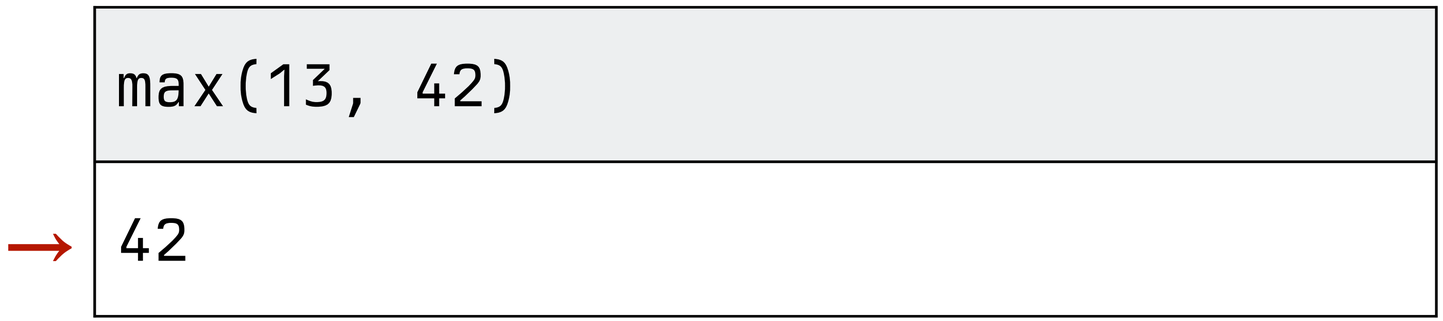

What function to call
f(42)


What function to call
Argument to the function
f(42)



"Call f on 42."


What function to call
First argument
Second argument
max(13, 42)


max(13, 42)
42

Sometimes the same computation can be done with an operator or a function, e.g.,

```
10 ** 2
```

→ `100`

```
pow(10, 2)
```

→ `100`

Call expression format	*Return value*
`min(`*arg*$_1$`, `*arg*$_2$`, ...)`	Minimum value out of two or more arguments
`max(`*arg*$_1$`, `*arg*$_2$`, ...)`	Maximum value out of two or more arguments
`abs(`*arg*`)`	Absolute value of the provided argument
`pow(`*base*`, `*exponent*`)`	Base number raised to the exponent
`round(`*number*`, `*precision*`)`	Round the number to the specified number of decimal places

Demo in notebook

Example: *Incidence of tuberculosis*

Morbidity and Mortality Weekly Report (*MMWR*)

Search

Tuberculosis — United States, 2021

Weekly / March 25, 2022 / 71(12);441–446

Print

Thomas D. Filardo, MD[1,2]; Pei-Jean Feng, MPH[2]; Robert H. Pratt[2]; Sandy F. Price[2]; Julie L. Self, PhD[2] (VIEW AUTHOR AFFILIATIONS)

View suggested citation

Summary

What is already known about this topic?

The number of reported U.S. tuberculosis (TB) cases decreased sharply in 2020, possibly related to multiple factors associated with the COVID-19 pandemic.

What is added by this report?

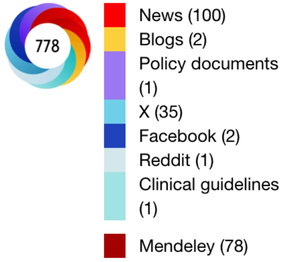
Reported TB incidence (cases per 100,000 persons) increased 9.4%, from 2.2 during 2020 to 2.4 during 2021 but was lower than incidence during 2019 (2.7). Increases occurred among both U.S.-born and non–U.S.-born persons.

What are the implications for public health practice?

Factors contributing to changes in reported TB during 2020–2021 likely include an actual reduction in TB incidence as well as delayed or missed TB diagnoses. Timely evaluation and treatment of TB and latent tuberculosis infection remain critical to achieving U.S. TB elimination.

Article Metrics

Altmetric:



50 Total citations

Morbidity and Mortality Weekly Report (*MMWR*)

Search

Tuberculosis — United States, 2021

Weekly / March 25, 2022 / 71(12);441–446

Print

Thomas D. Filardo, MD[1,2]; Pei-Jean Feng, MPH[2]; Robert H. Pratt[2]; Sandy F. Price[2]; Julie L. Self, PhD[2] (VIEW AUTHOR AFFILIATIONS)

View suggested citation

Summary

What is already known about this topic?

What is *incidence*?
Why use it here?

is (TB) cases decreased sharply in 2020, possibly related to VID-19 pandemic.

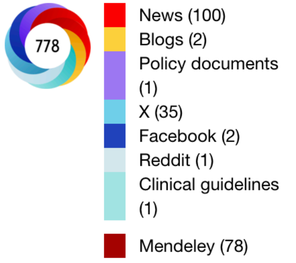
Reported TB incidence (cases per 100,000 persons) increased 9.4%, from 2.2 during 2020 to 2.4 during 2021 but was lower than incidence during 2019 (2.7). Increases occurred among both U.S.-born and non–U.S.-born persons.

What are the implications for public health practice?

Factors contributing to changes in reported TB during 2020–2021 likely include an actual reduction in TB incidence as well as delayed or missed TB diagnoses. Timely evaluation and treatment of TB and latent tuberculosis infection remain critical to achieving U.S. TB elimination.

Article Metrics

Altmetric:



50 Total citations

TABLE 1. Tuberculosis disease case counts and incidence, by U.S. state — 50 states and the District of Columbia, 2019–2021

Return

U.S. jurisdiction	No. of TB cases*			TB incidence†		
	2019	2020	2021	2019	2020	2021
Total	**8,900**	**7,173**	**7,860**	**2.71**	**2.16**	**2.37**
Alabama	87	72	92	1.77	1.43	1.83
		58	58	7.91	7.92	7.92
	3	136	129	2.51	1.89	1.77
		59	69	2.12	1.96	2.28
	1	1,706	1,750	5.35	4.32	4.46
Colorado	66	52	58	1.15	0.90	1.00
Connecticut	67	54	54	1.88	1.50	1.50
Delaware	18	17	43	1.84	1.71	4.29
District of Columbia	24	19	19	3.39	2.75	2.84

Let's use Python to validate these values. We'll check the 2020 and 2021 incidence for the US as a whole.

TABLE 1. Tuberculosis disease case counts and incidence, by U.S. state — 50 states and the District of Columbia, 2019–2021

Return

U.S. jurisdiction	No. of TB cases*			TB incidence†		
	2019	2020	2021	2019	2020	2021
Total	**8,900**	**7,173**	**7,860**	**2.71**	**2.16**	**2.37**
Alabama	87	72	92	1.77	1.43	1.83
		58	58	7.91	7.92	7.92
	3	136	129	2.51	1.89	1.77
		59	69	2.12	1.96	2.28
	1	1,706	1,750	5.35	4.32	4.46
Colorado	66	52	58	1.15	0.90	1.00
Connecticut	67	54	54	1.88	1.50	1.50
Delaware	18	17	43	1.84	1.71	4.29
District of Columbia	24	19	19	3.39	2.75	2.84

† Cases per 100,000 persons using midyear population estimates from the U.S. Census Bureau. 2019 population estimates are based on the 2010 U.S. Census. 2020 and 2021 population estimates are based on the 2020 U.S. Census. https://www.census.gov/programs-surveys/popest/data/tables.html

Let's use Python to validate these values. We'll check the 2020 and 2021 incidence for the US as a whole.

Demo in notebook

Types of values

Numbers

```
2 + 3
```

→ `5`

```
10 / 2
```

→ `5.0`

Why is Python displaying the same number two ways?

What we saw were two different *data types* used for numbers in Pythons:

Integers

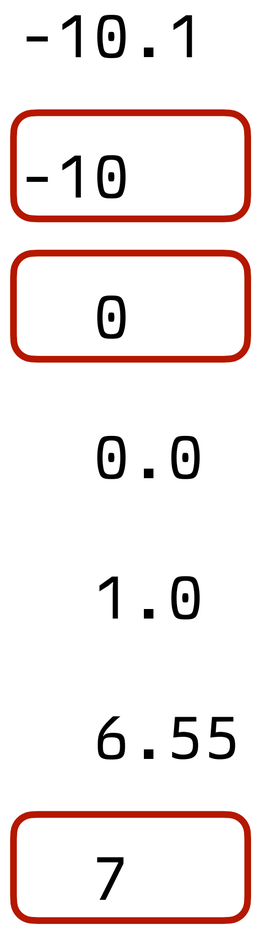
Floating-point numbers

```
-10.1

-10

  0

  0.0

  1.0

  6.55

  7
```

What we saw were two different *data types* used for numbers in Pythons:

Integers are whole numbers

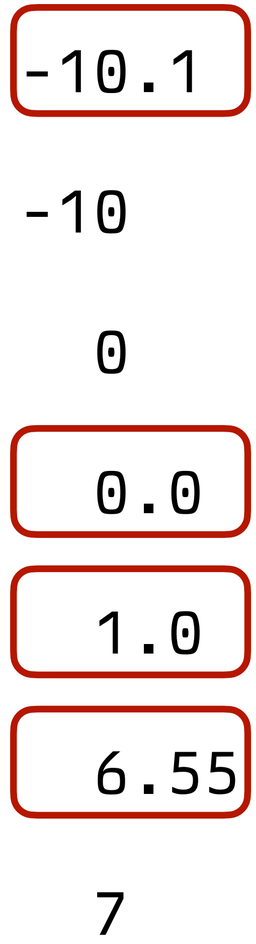
Floating-point numbers



What we saw were two different *data types* used for numbers in Pythons:

Integers are whole numbers

Floating-point numbers have a decimal point



Adding subtracting, and multiplying integers always gives you another integer.

But if there's any floating-point number, the result is another float – even if the decimal part is zero!

```
3 + (2 ** 9) - 15 * 14 + 1    → 306
3 + (2 ** 9) - 15 * 14 + 1.0  → 306.0
```

Division (/) *always* results in a float since the result isn't guaranteed to be a whole number:

```
15 / 3 → 5.0
```

When performing calculations with floating-point numbers, you'll sometimes see a small amount of *error* in the result:

```
0.1 + 0.1
```

→ `0.2`

```
0.1 + 0.2 + 0.3
```

→ `0.6000000000000001`

This is a consequence of how Python internally represents floats. There's nothing for you to do about it except be aware!

If you're curious about the details of that representation, you can read more about it.

String values

Text *strings* are values consisting of a sequence of characters (letters, numbers, punctuation, emoji, etc.):

```
"Poughkeepsie"
'New York'
```

Strings can be written between either single or double quotes.

You can *concatenate* (combine) strings using the **+** operator:

```
"Hello," + " " + "world!"
```

→ `"Hello, world!"`

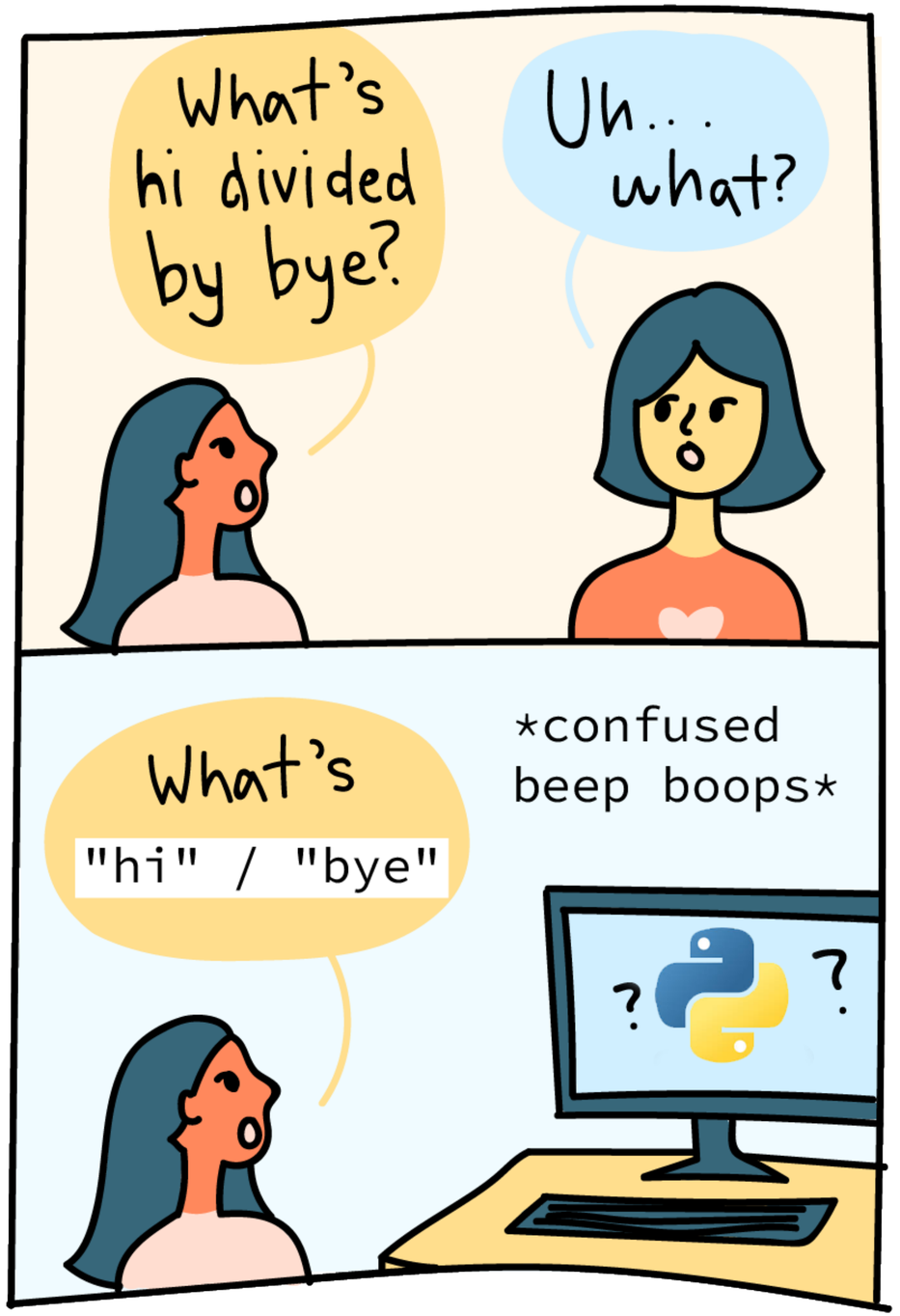
And you can use the **`len`** function to ask for the *length* of a string – how many characters are in it:

```
len("Hello")
```

→ `5`

Working with different types of values

Operations may only work on certain types of data!



The same operator or function may work differently when it's given different types of data as input

```
3 + 4
```

```
"3" + "4"
```

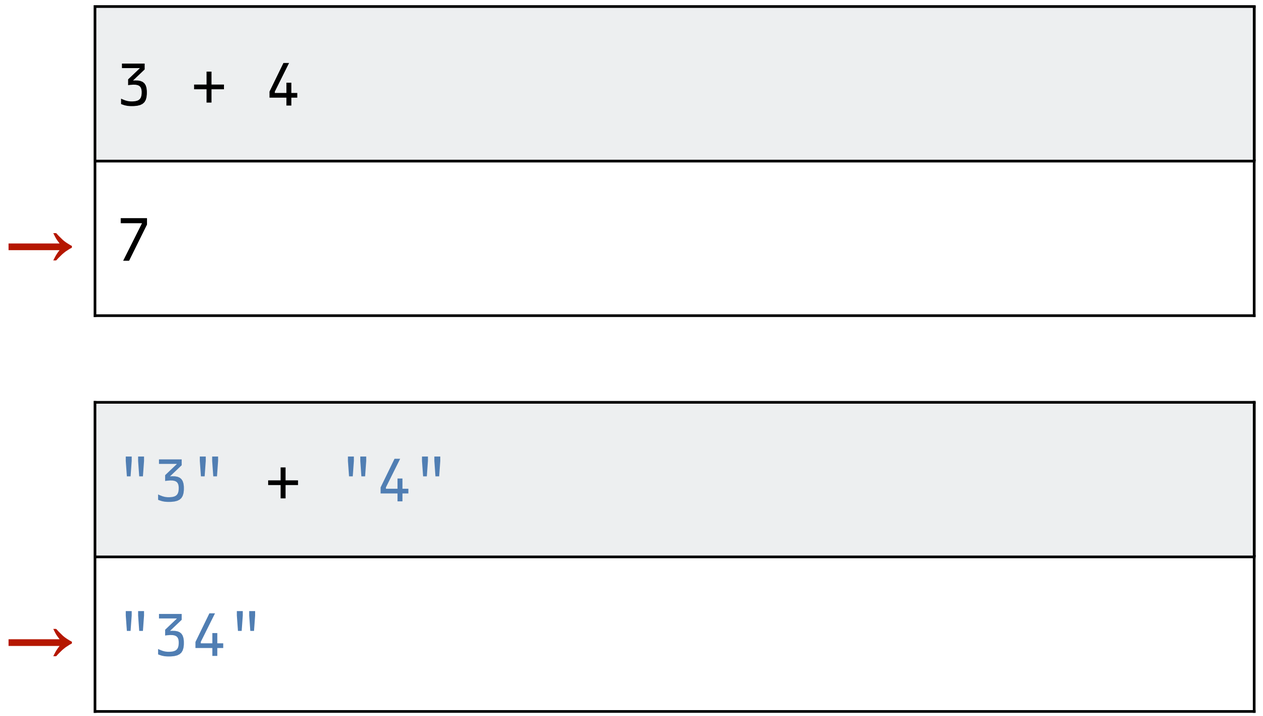
The same operator or function may work differently when it's given different types of data as input

```
3 + 4
```

→ `7`

```
"3" + "4"
```

The same operator or function may work differently when it's given different types of data as input



The same operator or function may work differently when it's given different types of data as input

```
3 * 4
```

```
"3" * 4
```

The same operator or function may work differently when it's given different types of data as input

```
3 * 4
```

→ 12

```
"3" * 4
```

The same operator or function may work differently when it's given different types of data as input

```
3 * 4
```

→ 12

```
"3" * 4
```

→ "3333"

The same operator or function may work differently when it's given different types of data as input

```
max(3, 4)
```

```
max("three", "four")
```

The same operator or function may work differently when it's given different types of data as input

```
max(3, 4)
```

→ `4`

```
max("three", "four")
```

The same operator or function may work differently when it's given different types of data as input

```
max(3, 4)
```

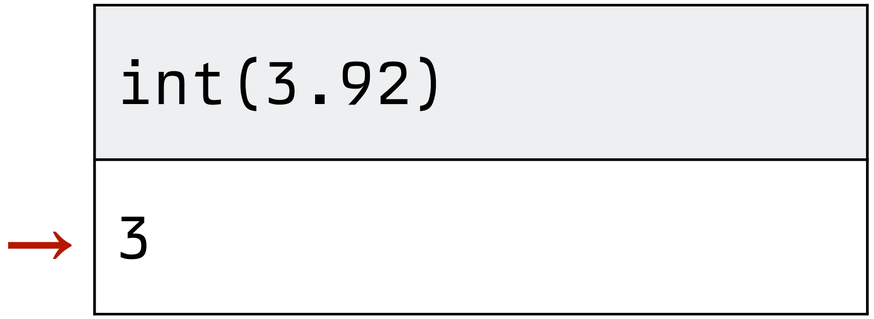
→ `4`

```
max("three", "four")
```

→ `"three"`

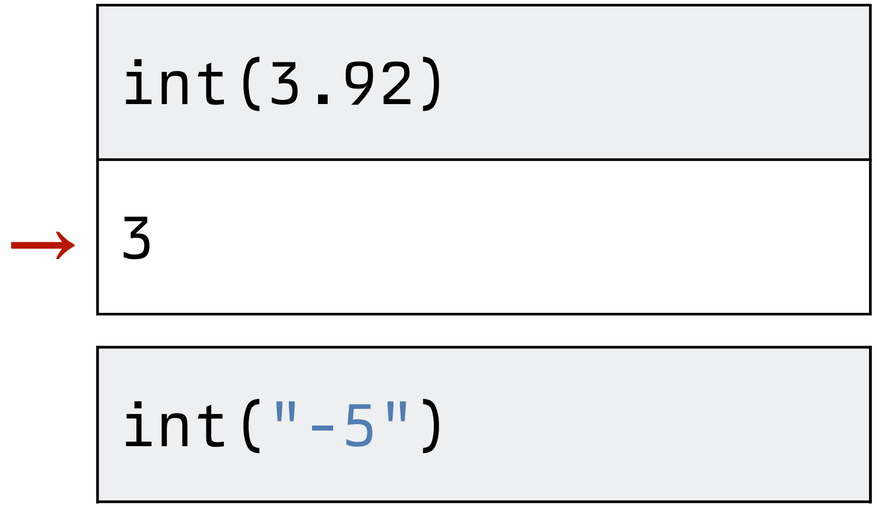
Prest-o change-o

When it makes sense, we can *typecast* – convert values between data types.



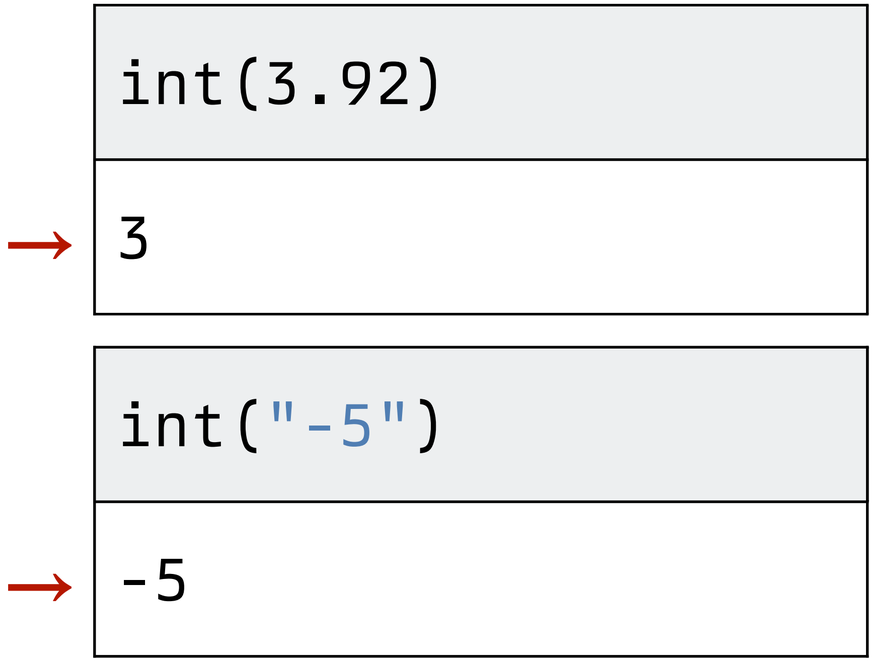
Prest-o change-o

When it makes sense, we can *typecast* – convert values between data types.



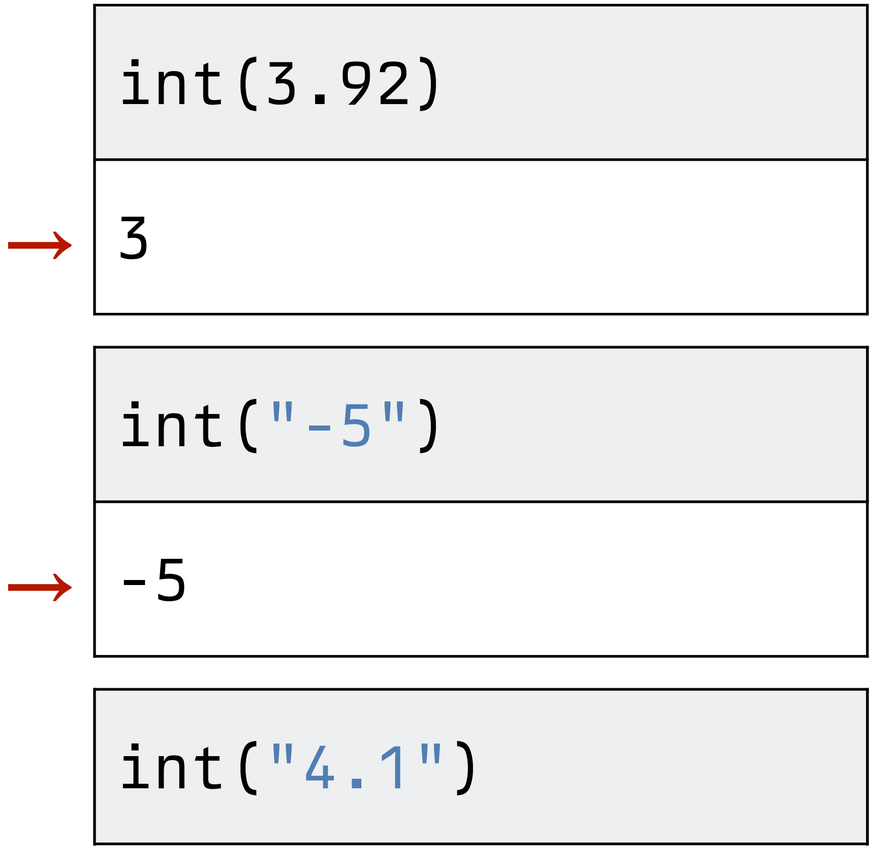
Prest-o change-o

When it makes sense, we can *typecast* – convert values between data types.



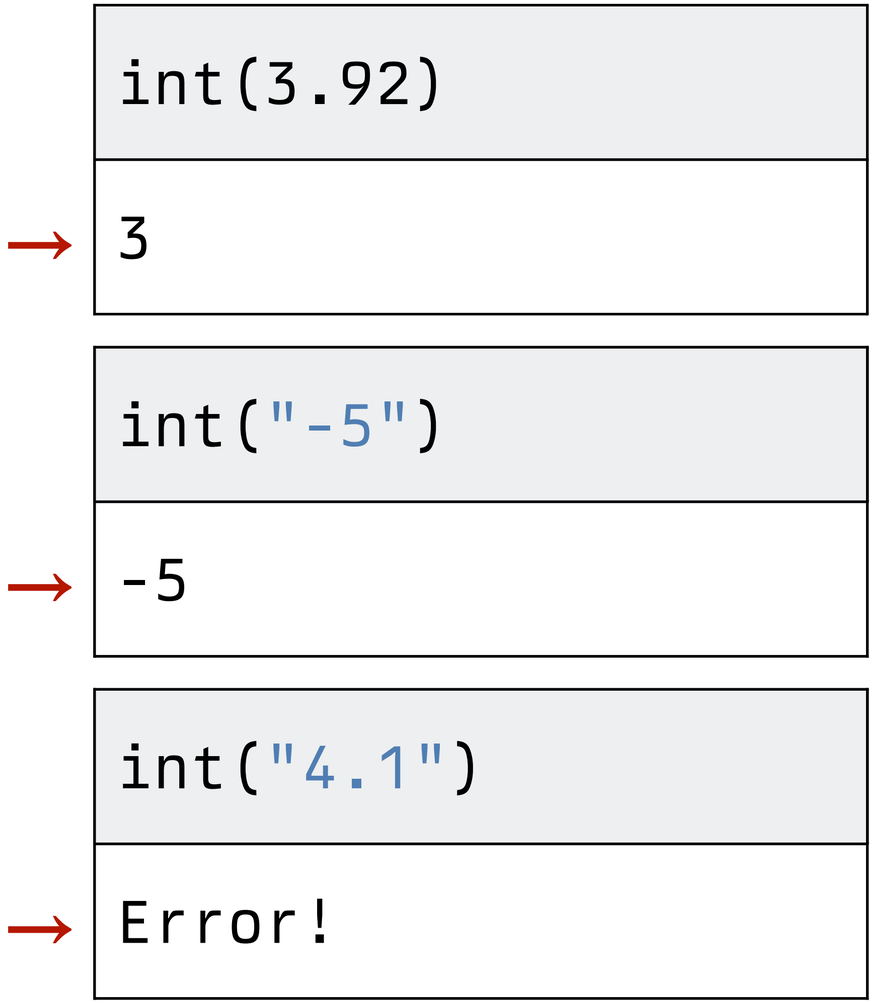
Prest-o change-o

When it makes sense, we can *typecast* – convert values between data types.



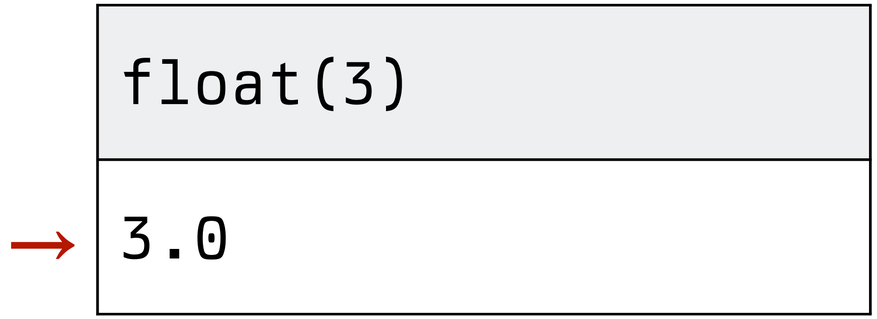
Prest-o change-o

When it makes sense, we can *typecast* – convert values between data types.



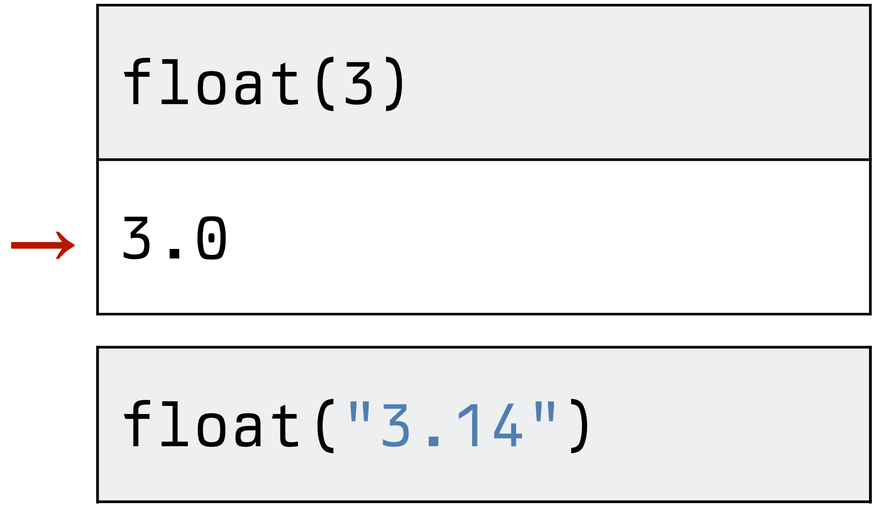
Prest-o change-o

When it makes sense, we can *typecast* – convert values between data types.



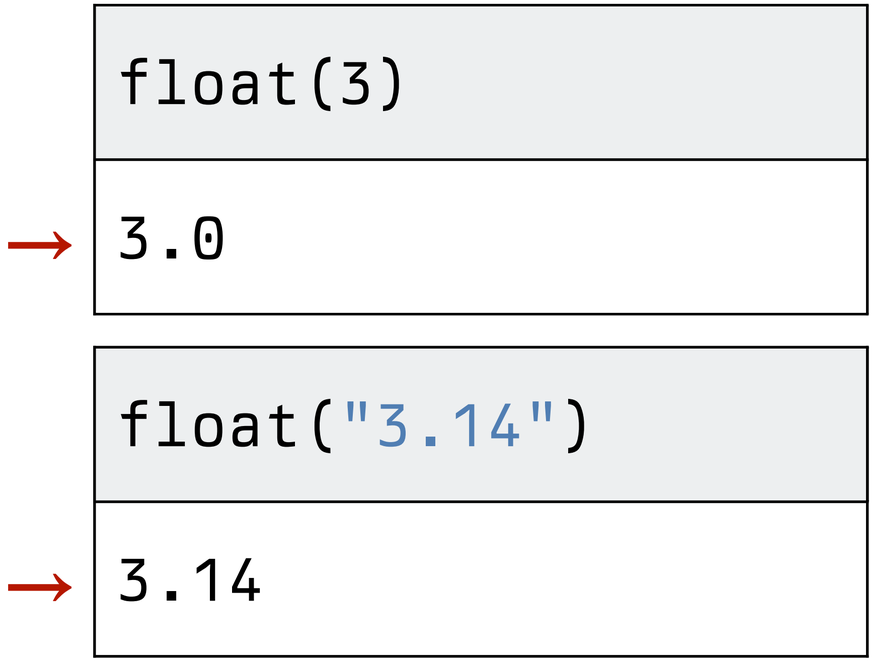
Prest-o change-o

When it makes sense, we can *typecast* – convert values between data types.



Prest-o change-o

When it makes sense, we can *typecast* – convert values between data types.



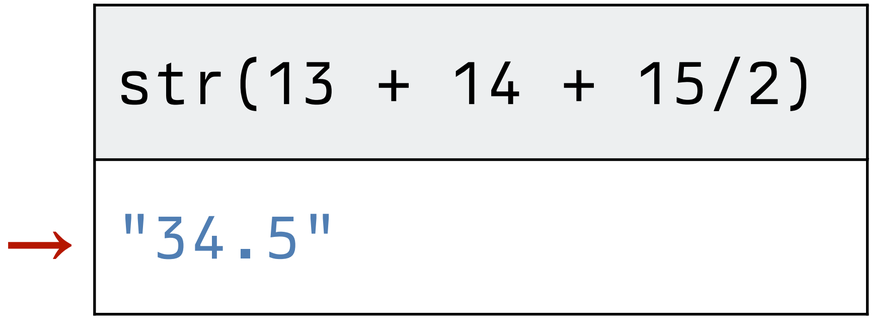
Prest-o change-o

When it makes sense, we can *typecast* – convert values between data types.

```
str(13 + 14 + 15/2)
```

Prest-o change-o

When it makes sense, we can *typecast* – convert values between data types.



Evaluation

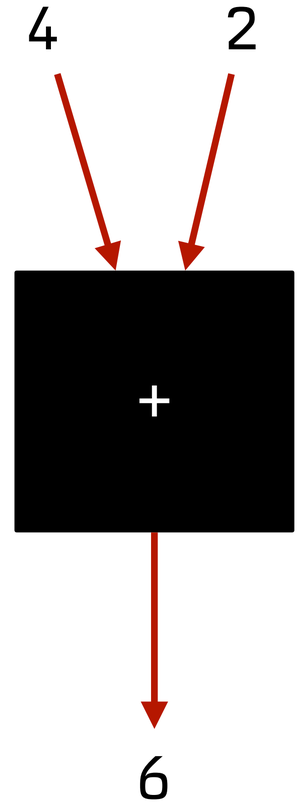
How does something like `(4 + 2) / 3` work?

What is the operator `/` dividing?

Shouldn't `/` expect two numbers?

Even though `(4 + 2)` isn't a number, it's an expression that *evaluates* to a number.

This works for all data types, not just numbers!



When we write complex expressions, Python evaluates them from the inside out:

```
7 + (6 / (1 + 1))
```

When we write complex expressions, Python evaluates them from the inside out:

```
7 + (6 / (1 + 1))
```

→

```
7 + (6 / 2)
```

When we write complex expressions, Python evaluates them from the inside out:

```
  7 + (6 / (1 + 1))
→ 7 + (6 / 2)
→ 7 + 3
```

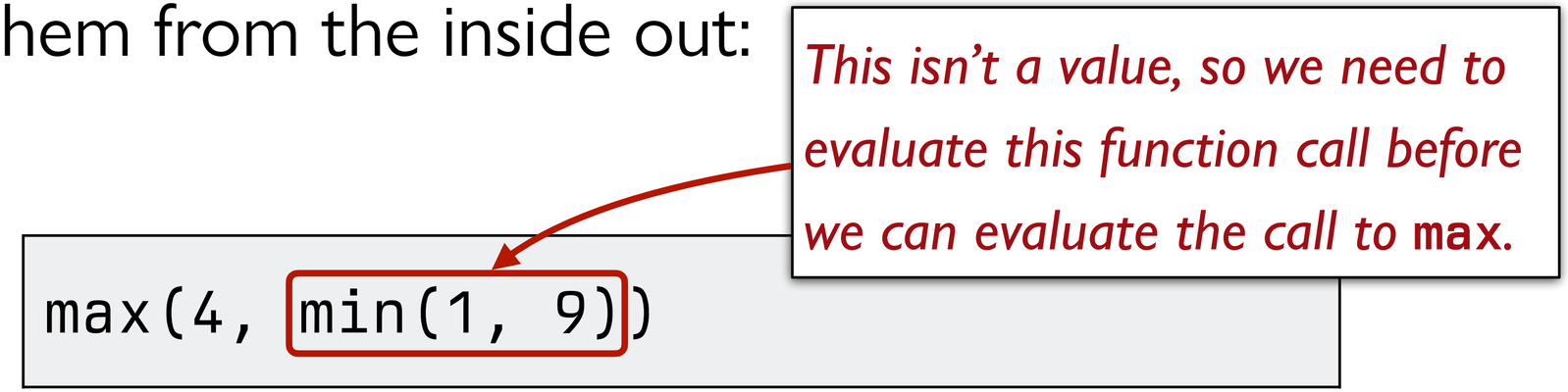
When we write complex expressions, Python evaluates them from the inside out:

```
  7 + (6 / (1 + 1))
→ 7 + (6 / 2)
→ 7 + 3
→ 10
```

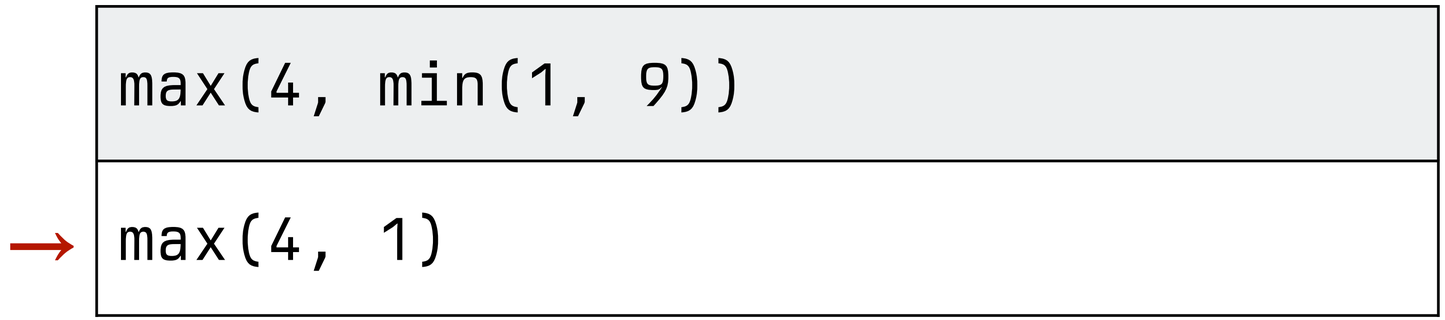
When we write complex expressions, Python evaluates them from the inside out:

```
max(4, min(1, 9))
```

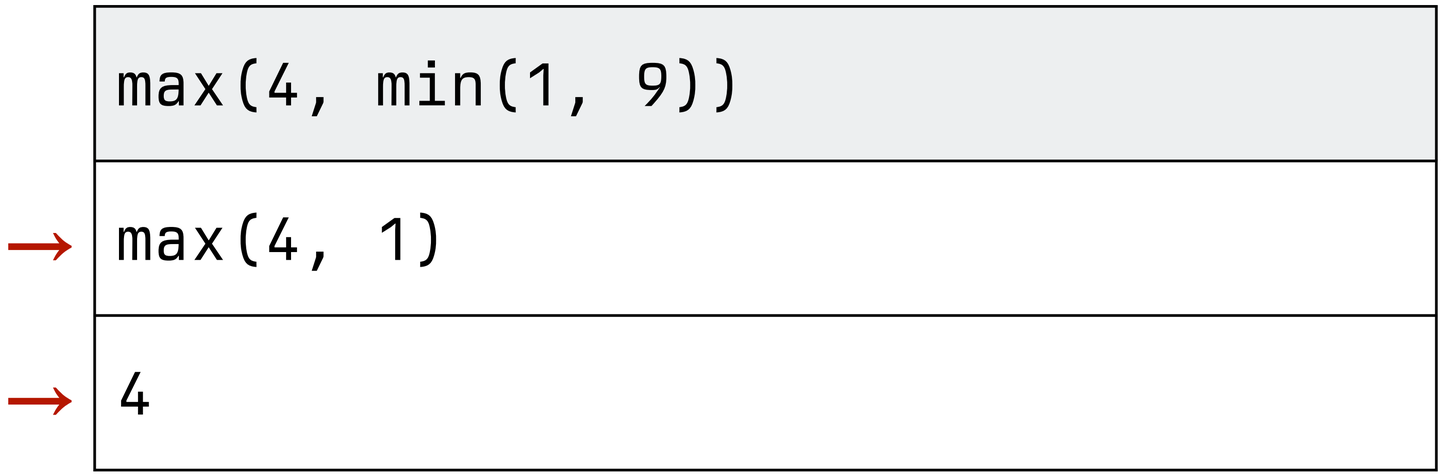
When we write complex expressions, Python evaluates them from the inside out:



When we write complex expressions, Python evaluates them from the inside out:



When we write complex expressions, Python evaluates them from the inside out:



We can nest as many function calls as we want!

```
min(abs(max(-1, -2, -3, min(4, -2))), max(5, 100))
```

We can nest as many function calls as we want!

```
min(abs(max(-1, -2, -3, min(4, -2))), max(5, 100))
```

We can nest as many function calls as we want!

```
min(abs(max(-1, -2, -3, min(4, -2))), max(5, 100))
```

→

```
min(abs(max(-1, -2, -3, min(4, -2))), 100)
```

We can nest as many function calls as we want!

```
min(abs(max(-1, -2, -3, min(4, -2))), max(5, 100))
```

→

```
min(abs(max(-1, -2, -3, min(4, -2))), 100)
```

We can nest as many function calls as we want!

```
min(abs(max(-1, -2, -3, min(4, -2))), max(5, 100))
```

→

```
min(abs(max(-1, -2, -3, min(4, -2))), 100)
```

→

```
min(abs(max(-1, -2, -3, -2)), 100)
```

We can nest as many function calls as we want!

```
min(abs(max(-1, -2, -3, min(4, -2))), max(5, 100))
```

→

```
min(abs(max(-1, -2, -3, min(4, -2))), 100)
```

→

```
min(abs(max(-1, -2, -3, -2)), 100)
```

We can nest as many function calls as we want!

```
min(abs(max(-1, -2, -3, min(4, -2))), max(5, 100))
```

→

```
min(abs(max(-1, -2, -3, min(4, -2))), 100)
```

→

```
min(abs(max(-1, -2, -3, -2)), 100)
```

→

```
min(abs(-1), 100)
```

We can nest as many function calls as we want!

```
min(abs(max(-1, -2, -3, min(4, -2))), max(5, 100))
```

→

```
min(abs(max(-1, -2, -3, min(4, -2))), 100)
```

→

```
min(abs(max(-1, -2, -3, -2)), 100)
```

→

```
min(abs(-1), 100)
```

We can nest as many function calls as we want!

```
min(abs(max(-1, -2, -3, min(4, -2))), max(5, 100))
→ min(abs(max(-1, -2, -3, min(4, -2))), 100)
→ min(abs(max(-1, -2, -3, -2)), 100)
→ min(abs(-1), 100)
→ min(1, 100)
```

We can nest as many function calls as we want!

```
min(abs(max(-1, -2, -3, min(4, -2))), max(5, 100))
```

→

```
min(abs(max(-1, -2, -3, min(4, -2))), 100)
```

→

```
min(abs(max(-1, -2, -3, -2)), 100)
```

→

```
min(abs(-1), 100)
```

→

```
min(1, 100)
```

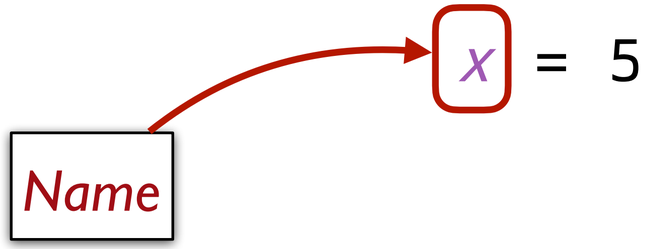
We can nest as many function calls as we want!

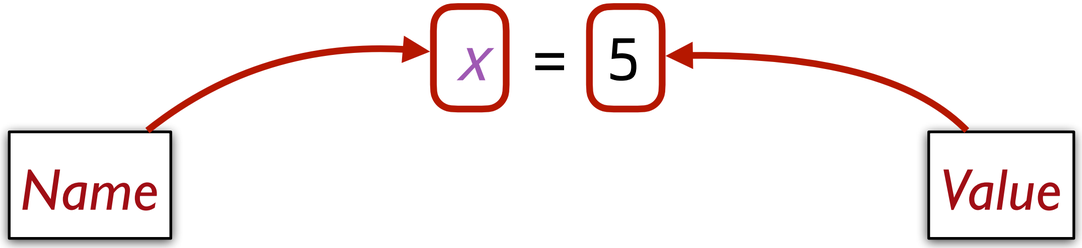
```
min(abs(max(-1, -2, -3, min(4, -2))), max(5, 100))
→ min(abs(max(-1, -2, -3, min(4, -2))), 100)
→ min(abs(max(-1, -2, -3, -2)), 100)
→ min(abs(-1), 100)
→ min(1, 100)
→ 1
```

Names

Defining names

$$x = 5$$


x
=
5
Name


x
=
5
Name
Value

$x = 5$

The name **x** *is bound to the value* **5**, *like putting a baggage tag on a suitcase.*

```
x = 5

y = 1 + 2 * 3 - 8 / 2
```

First Python evaluates the right-hand expression…

```
x = 5

y = 1 + 2 * 3 - 8 / 2
→ y = 1 + 6     - 8 / 2
```

First Python evaluates the right-hand expression…

```
x = 5

  y = 1 + 2 * 3 - 8 / 2
→ y = 1 + 6     - 8 / 2
→ y = 7         - 4
```

First Python evaluates the right-hand expression…

```
x = 5

  y = 1 + 2 * 3 - 8 / 2
→ y = 1 + 6     - 8 / 2
→ y = 7         - 4
→ y = 3
```

First Python evaluates the right-hand expression…

```
x = 5

y = 1 + 2 * 3 - 8 / 2
→ y = 1 + 6     - 8 / 2
→ y = 7         - 4
→ y = 3
```

First Python evaluates the right-hand expression…

…then it binds the name y to the resulting value.

Several names may have the same value:

```
seven = 7
seven
```

```
7
```

```
septem = 7
septem
```

```
7
```

Assignment statements are *not* mathematical equations.

If you write

```
3 = x
```

Python gives a syntax error because it thinks you're trying to redefine what "3" means.

Name examples

$x = 5$

```
x = 5
```

There's no output from assigning a name to a value.

```
x = 5
```

Directory

Name	***Value***
x	5

There's no output from assigning a name to a value.

It has the side effect of associating the name with the value in the program directory.

x = 5

x

Directory

Name	***Value***
x	5

```
x = 5
```

```
x
```

Directory

Name	***Value***
x	5

When you use the name later, Python looks it up in the directory and substitutes the value it finds.

x = 5

x

→ 5

Directory

Name	***Value***
x	5

When you use the name later, Python looks it up in the directory and substitutes the value it finds.

```
fname = "Grace"
```

Directory

Name	***Value***

```
fname = "Grace"
```

Directory

Name	***Value***
fname	"Grace"

```
fname = "Grace"
```

```
lname = "Hopper"
```

Directory

Name	***Value***
fname	"Grace"

```
fname = "Grace"
```

```
lname = "Hopper"
```

Directory

Name	***Value***
fname	"Grace"
lname	"Hopper"

```
fname = "Grace"
```

```
lname = "Hopper"
```

```
fname + " " + lname
```

Directory

Name	***Value***
fname	"Grace"
lname	"Hopper"

```
fname = "Grace"
```

```
lname = "Hopper"
```

```
fname + " " + lname
```

→

```
"Grace" + " " + lname
```

Directory

Name	***Value***
fname	"Grace"
lname	"Hopper"

```
fname = "Grace"
```

```
lname = "Hopper"
```

```
fname + " " + lname
```

→ `"Grace" + " " + lname`

→ `"Grace " + lname`

Directory

Name	***Value***
fname	"Grace"
lname	"Hopper"

```
fname = "Grace"
```

```
lname = "Hopper"
```

```
fname + " " + lname
```

→ `"Grace" + " " + lname`

→ `"Grace " + lname`

→ `"Grace " + "Hopper"`

Directory

Name	***Value***
fname	"Grace"
lname	"Hopper"

```
fname = "Grace"
```

```
lname = "Hopper"
```

```
fname + " " + lname
```

→ `"Grace" + " " + lname`

→ `"Grace " + lname`

→ `"Grace " + "Hopper"`

→ `"Grace Hopper"`

Directory

Name	***Value***
fname	"Grace"
lname	"Hopper"

Working with names

A name can only be bound to a single value at one time.

A name can only be bound to a single value at one time.

$x = 2$

A name can only be bound to a single value at one time.

$x = 2$

A name can only be bound to a single value at one time.

```
x = 2
x = x + 1
```

A name can only be bound to a single value at one time.

```
x = 2
x = x + 1
```

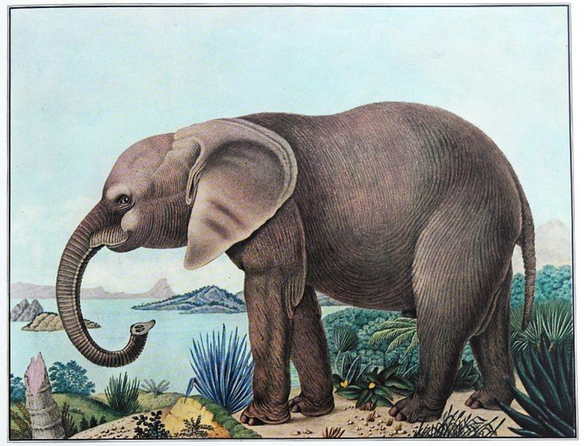
Names must be given a value before being used.

```
new_name
```

```
NameError
Traceback (most recent call last)
<ipython-input-1-9d86db7a2999> in <cell line: 1>()
----> 1 new_name

NameError: name 'new_name' is not defined
```

Jupyter memory model

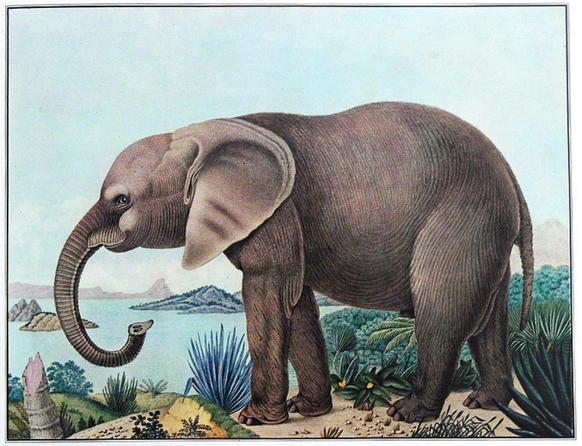
Pretend your notebook has a brain.



Jupyter memory model

Pretend your notebook has a brain.

Every time you run a cell with an assignment statement, it remembers that name–value binding.

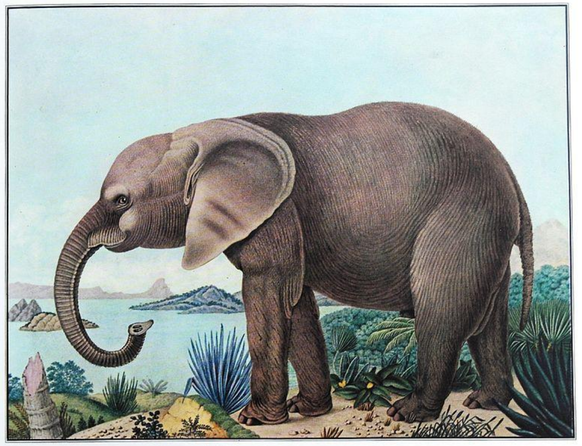


Jupyter memory model

Pretend your notebook has a brain.

Every time you run a cell with an assignment statement, it remembers that name–value binding.

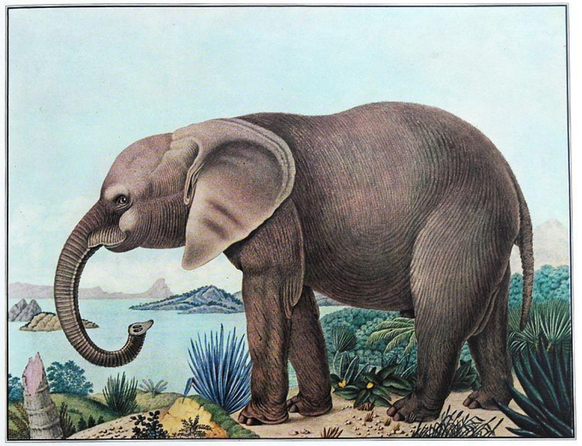
It will remember all name–value mappings as long *as the current session is running*, no matter how many cells you create.



Jupyter memory model

Pretend your notebook has a brain.

> Every time you run a cell with an assignment statement, it remembers that name–value binding.
>
> It will remember all name–value mappings as long *as the current session is running*, no matter how many cells you create.



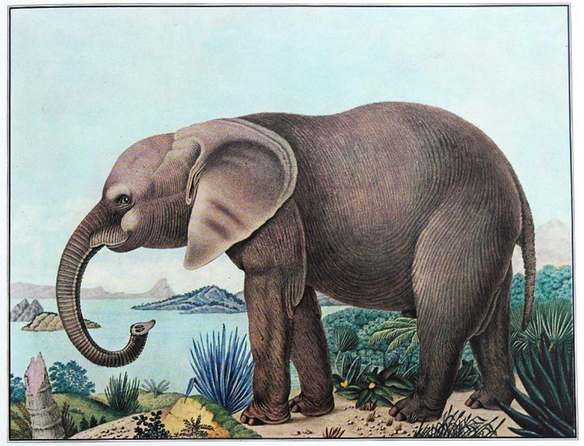
However, when you open a notebook for the first time in a few hours, your previous session will likely have ended, and Jupyter's brain won't remember anything.

Jupyter memory model

Pretend your notebook has a brain.

Every time you run a cell with an assignment statement, it remembers that name–value binding.

It will remember all name–value mappings as long *as the current session is running*, no matter how many cells you create.



However, when you open a notebook for the first time in a few hours, your previous session will likely have ended, and Jupyter's brain won't remember anything.

You'll need to re-run all of your cells.

Don’t delete cells defining names you want to use.

Don’t use names *above* the cell with the assignment definition.

Notebooks should be a paper trail. Each cell is a record of what you’ve done so far.

What's in a name?

If you're ever unsure of the value bound to a name, you can simply create a new cell, type the name, and run the cell.

Python has *built-in* names, including functions like `min`, `max`, and `pow`.

Python will let you re-assign some of these built-in names, even though you probably shouldn't!

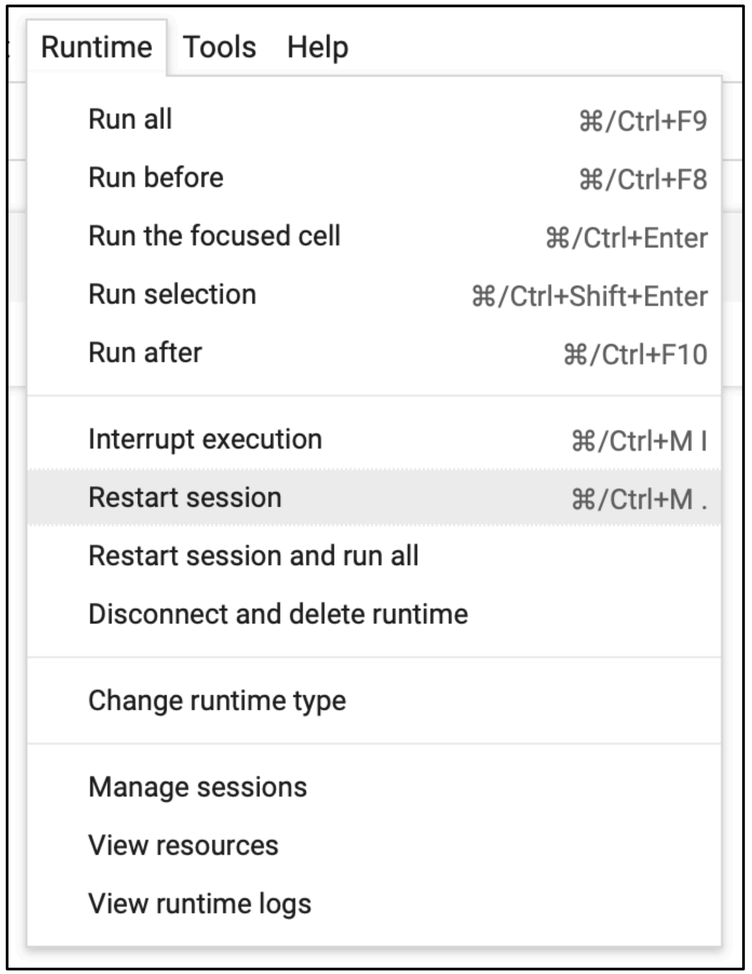
```
max = 9
max(2, 3)
```

```
TypeError: 'int' object is not callable
```

You broke Python. What now?

If you want to restore names to their default values, do this:

1. Save your notebook
2. Restart your session



There are also some *reserved names*, e.g.,

```
import
None
True
False
```

These are so important to Python that reassigning them would be a big problem, so it won't let you do it.

Concept check

We can define the names

```
width = 400
height = 600
```

Now if we write

```
width * height
```

it gets evaluated:

What if we use another name?

```
width = 400
height = 600
area = width * height
```

Does Python associate the name **`area`** with the expression **`width * height`** or with the number **`240000`**?

Writing code for people to read

“Programs must be written for people to read, and only incidentally for machines to execute.”

Hal Abelson & Gerald Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, 1979

Choosing good names

Names are arbitrary

This is silly, but legal:

```
five = 6
five
```

```
6
```

```
six = 5
six
```

```
5
```

Names in Python are case-sensitive.

So,

```
Cat
CAT
cat
cAT
```

are all distinct names, which can have different associated values – but doing this is a bad idea because it's confusing!

They look similar – but they're all distinct!

Python is pretty flexible about what names can look like:

- 👍 `how_are_you`
- 👍 `my_AGE_is_22`
- 👍 `NETFLIXPASSWORD`

But it doesn't allow hyphens or other punctuation – only underscores are allowed:

- 👎 `this-is-bad`
- 👎 `worse!`
- 👎 `no&*way`

While names can include a number, like

👍 `pi_r_2`

They can't *start* with a number:

👎 `2_pi_r`

Every programming language also has its own *conventions* for names.

In standard Python, names are usually lowercase with words joined by underscores, e.g.,

```
this_is_a_good_name
```

```
thisMakesPythonCRY
```

Names are important!

Can you guess what this code does?

```
y = (x + 459.67) * 5/9
```

Names are important!

Can you guess what this code does?

```
y = (x + 459.67) * 5/9
```

```
temp_kelvin = (temp_celsius + 459.67) * 5/9
```

Choose names that are concise but descriptive.

Good:

```
seconds_per_hour = 60 * 60
hours_per_year = 24 * 365
seconds_per_year = seconds_per_hour * hours_per_year
```

Not so good:

```
i_love_chocolate = 60 * 60 * 24 * 365
```

Comments

Comments are used to explain what code does.

Good programmers write code that is *self-evident*
and use comments only where necessary.

```
7173 / (331501080 / 100000)
```

```
7173 / (331501080 / 100000)
```

```
7173 / (331501080 / 100000)
```

```
# 2020 US TB incidence
7173 / (331501080 / 100000)
```

```
7173 / (331501080 / 100000)
```

```
# 2020 US TB incidence
7173 / (331501080 / 100000)
```

```
7173 / (331501080 / 100000)
```

🤔

```
# 2020 US TB incidence
7173 / (331501080 / 100000)
```

😀

```
# 7171 ÷ (100,000 groups in 331,501,080 population)
7173 / (331501080 / 100000)
```

```
7173 / (331501080 / 100000)
```

```
# 2020 US TB incidence
7173 / (331501080 / 100000)
```

```
# 7171 ÷ (100,000 groups in 331,501,080 population)
7173 / (331501080 / 100000)
```

Girl on the Net
@girlonthenet@mastodon.social

Fun fact: the code which took Apollo 11 to the moon is available on github github.com/chrislgarry/Apollo-...

And if you look through it you'll see that - joyfully - it also includes original comments.

My absolute favourite thing about the Moon Code is that it includes comments like this: "TEMPORARY - I HOPE HOPE HOPE"

```
                TS      WCHPHASE
                TC      BANKCALL        # TEMPORARY, I HOPE HOPE HOPE
180             CADR    STOPRATE        # TEMPORARY, I HOPE HOPE HOPE
181             TC      DOWNFLAG        # PERMIT X-AXIS OVERRIDE
ALT             ADRES   XOVINFLG
```

Aug 30, 2024, 07:24 AM · · Web

Acknowledgments

This class incorporates material from:

Kathi Fisler, Brown University

Jason Waterman, Vassar College

Data 6, University of California, Berkeley (CC BY-NC-SA)