STATS 507
Data Analysis in Python

Week1.2: Data Types, Conditionals, Functions

Dr. Xian Zhang

Recap: 1st lecture

Read Syllabus on Canvas

Install Anaconda and Jupyter

Sign up for github and create your own stats 507 repo.

Slides and in-class practice files up before lecture.

- Highly encourage you to download them before class
- Take notes and run code files when I do
- Class will not be recorded.

Instructors

Xian Zhang, *xianz@umich.edu*

Ph.D., Software Engineer

https://sites.google.com/view/xian-zhang/home

My office hours:

Fridays 3:30 PM – 5:00 PM

Zoom: https://umich.zoom.us/j/8786375189

Graduate Student Instructors

Marc Brooks, *marcbr@umich.edu*

Statistics PhD student

Matthew McAnea, *mmcanear@umich.edu*

Statistics PhD student

GSI Office hours:

Wednesdays 10:00 AM – 11:30 AM
Angell Hall 219

Tuesdays 9:00 AM – 11:00 AM
Angell Hall 219

Getting help

	Canvas discussion board	GSI office hours	my office hours	email GSIs	email me
questions about homework	🙂 *	🙂		✗	✗
questions about lecture / slides / course topics		🙂	😃	✗	✗
questions / concerns about grading	✗	🙂	🙂 **	🙂	
personal matters; concerns about course; extended illness	✗		✓	✗	✓

* For questions about homework that cannot be asked without revealing a solution, please ask during GSI office hours rather than through Canvas.

** Please ask the GSIs first about homework grading; come to me if your concern is not resolved.

Recap: what is Python?

Python is a **dynamically typed**, **interpreted** programming language

Design philosophy: simple, readable

Dynamically typed

In many languages, when you declare a variable, you must specify the variable's **type** (e.g., int, double, Boolean, string). Python does not require this, the type of a variable is defined at **runtime**.

v.s. statically typed, flexible yet more error-prone

Interpreted

Some languages (e.g. C/C++ and Java) are compiled: we write code, from which we get a runnable grogram via **compilation**. In contrast, Python is **interpreted**: a program, called the **interpreter**, runs our code directly, line by line.

v.s compiled: simple yet slower

These programming language translators fall into two general categories: (1) interpreters and (2) compilers.

An interpreter reads the source code of the program as written by the programmer, parses the source code, and interprets the instructions on
Python is an interpreter and when we are running Python interactively, we can type a line of Python (a sentence) and Python processes it imm
and is ready for us to type another line of Python.

It is the nature of an interpreter to be able to have an interactive conversation as shown above. A compiler needs to be handed the entire pro
file, and then it runs a process to translate the high-level source code into machine language and then the compiler puts the resulting machin
into a file for later execution.

If you have a Windows system, often these executable machine language programs have a suffix of ".exe" or ".dll" which stand for "executable"
"dynamic link library" respectively. In Linux and Macintosh, there is no suffix that uniquely marks a file as executable.

Now let's write Python…

The Central Processing Unit (or CPU) is the part of the computer that is built to be obsessed with "what is next?" If your computer is rated a
Gigahertz, it means that the CPU will ask "What next?" three billion times per second. You are going to have to learn how to talk fast to kee
the CPU.

The Main Memory is used to store information that the CPU needs in a hurry. The main memory is nearly as fast as the CPU. But the inform
stored in the main memory vanishes when the computer is turned off.

The Secondary Memory is also used to store information, but it is much slower than the main memory. The advantage of the secondary me
that it can store information even when there is no power to the computer. Examples of secondary memory are disk drives or flash memory
found in USB sticks and portable music players).

The Input and Output Devices are simply our screen, keyboard, mouse, microphone, speaker, touchpad, etc. They are all of the ways we inte
the computer.

These days, most computers also have a Network Connection to retrieve information over a network. We can think of the network as a very
place to store and retrieve data that might not always be "up". So in a sense, the network is a slower and at times unreliable form of Second
Memory.

Python as a programming language

Aspects of language

Primitive constructs

- English: words -> sentences -> stories -> chapters -> books…
- Programming languages: numbers, strings, operators

  -> expressions-> functions-> modules -> apps …

Python as a programming language

Aspects of language -- Syntax

- English
  - “I stats 507.”
  - ”I love stats 507.”

Noun noun noun -> invalid syntax

Noun verb noun -> valid syntax

- Programming languages
  - “Hello World!” 100
  - “Hello World!” * 100

Object object -> invalid syntax

Object operator object -> valid syntax

Python as a programming language

Aspects of language -- Semantics (meaning)

The branch of linguistics and logic concerned with meaning

- English
  - ”I likes this class.”

Noun verb noun-> valid syntax
But semantic error

- Programming languages
  - “Hello World!” + 5

Object operator object -> valid syntax
But semantic error

Python as a programming language

Aspects of programming

A program is a sequence of definition and commands (recipe)…programs define and manipulate **data objects**

Unlike English, program can only have one meaning

- The chicken is ready to eat…

Where things could go wrong in Python…

Syntactic errors

- Common and easily caught
- Misspelled words, extra colons
- incorrect indentation, unmatched (){}[]…

Semantics errors

- Type mismatches…
- Use a variable before it is defined…

More on this later…

Run time & logical errors…

- Crashes
- Run-time error, programs run forever
- Generates the wrong answer…

1. Data Types in Python
2. Conditionals
3. Functions

Why do we need data types?

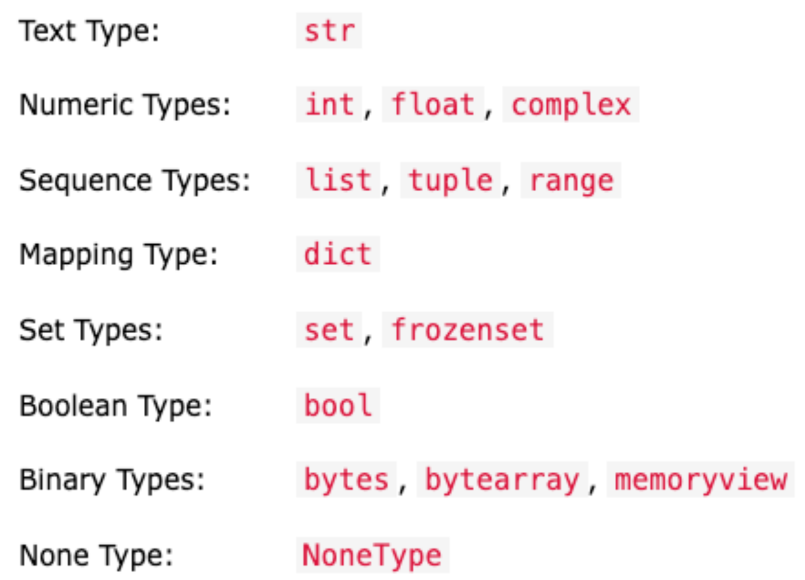
Different object can **represent** different concepts.

ANY object has a type that defines what kind of operations programs can do to them

- 30
  - Is a number
  - Can Add/sub/mul/div…
- “Stats 507”
  - Is a string (a sequence of characters)
  - Can get a substring, but can not div by a number…

Object data types in Python

- Built-in
  - Scalar (can not be subdivided)
    - Number
    - Truth, False
  - Non-scalar
    - Lists
    - Strings
    - Dictionaries
    - …
- Custom object
  - Tree
  - Graph
  - …



Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`
None Type:	`NoneType`

Ref: https://www.w3schools.com/python/python_datatypes.asp

Scalar Objects in Python

- int, -- represent integers, ex: 507
- float, -- represent real numbers, ex: 3.1415, 2.0
- bool, -- represent Boolean values, ex: True, False
- NoneType -- special and has one value, None

- Can use type() to see the type of any object

Let’s try it in Jupyter…

Dynamically typed one more time

Dynamically typed

In many languages, when you declare a variable, you must specify the variable's **type** (e.g., int, double, Boolean, string). Python does not require this, the type of a variable is defined at **runtime**.

v.s. statically typed, flexible yet more error-prone

Unlike some languages (e.g. C, C++, Java), **you don't tell Python the type of a variable** when you declare it.

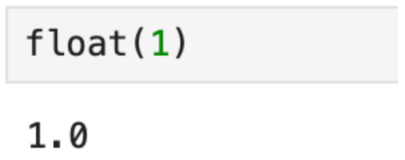
This is often called "**duck typing**".

If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.
https://en.wikipedia.org/wiki/Duck_test

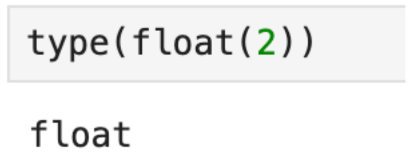
Pros and cons?

Type Conversion (Casting)

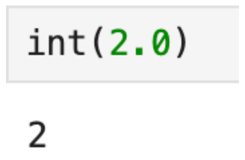
We can cast objects to different type…



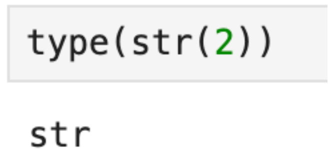
```
float(1)

1.0
```



```
type(float(2))

float
```

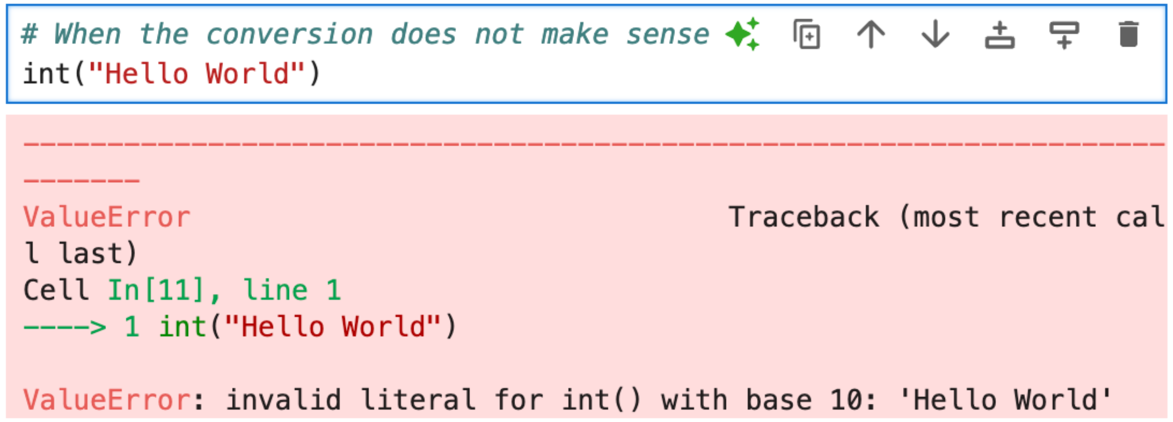


```
int(2.0)

2
```



```
type(str(2))

str
```

Not all conversions make sense…

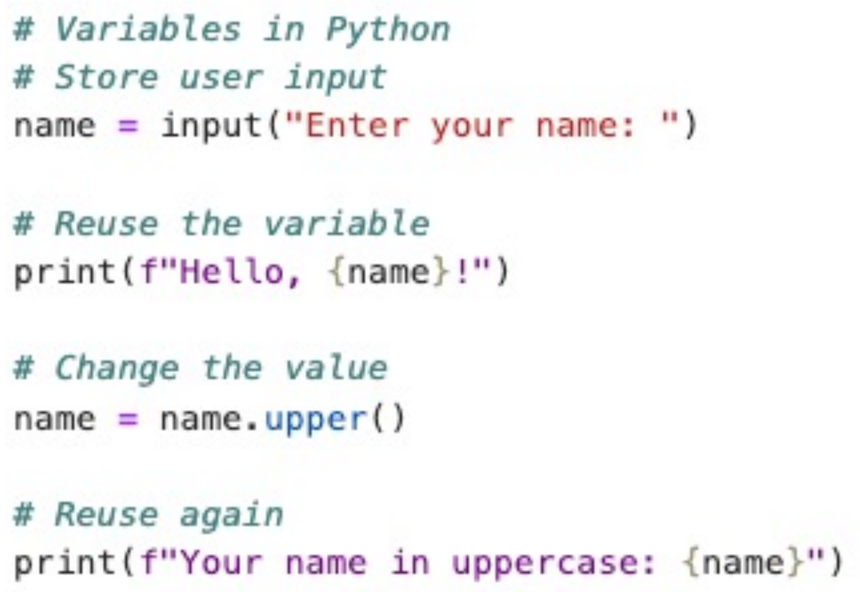


```
# When the conversion does not make sense
int("Hello World")

---------------------------------------------------------------------------
--------
ValueError                                Traceback (most recent cal
l last)
Cell In[11], line 1
----> 1 int("Hello World")

ValueError: invalid literal for int() with base 10: 'Hello World'
```

Variable in Python

Variable is a name that **refers** to a **value**.

Assign a value to a variable via assignment operator "="

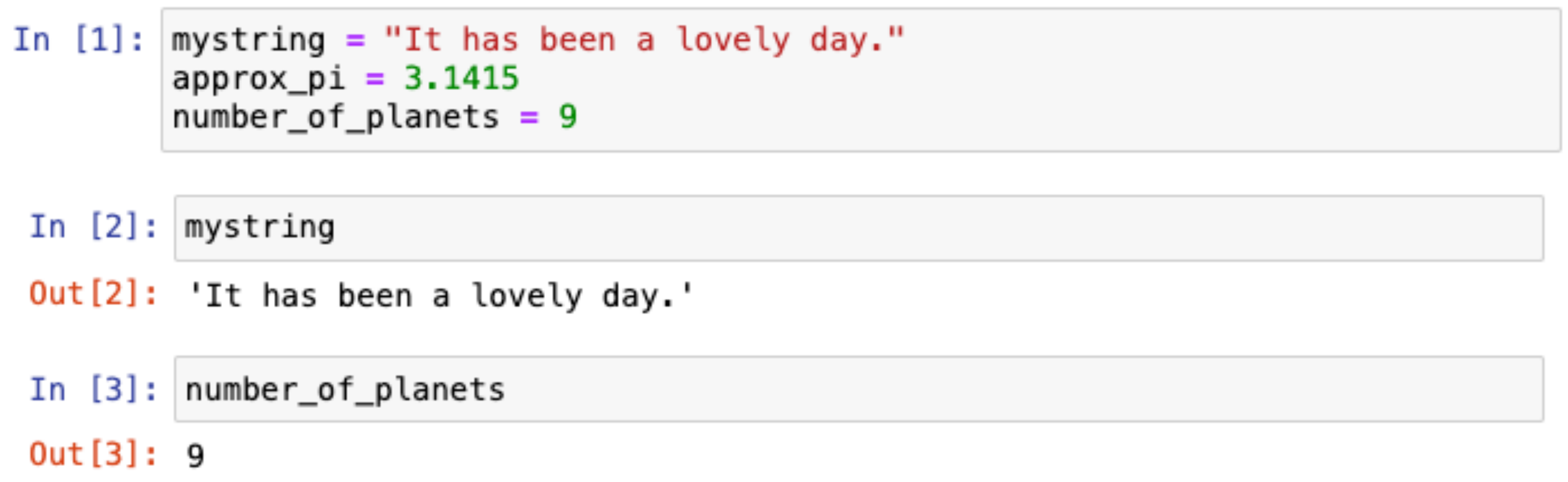


```python
# Variables in Python
# Store user input
name = input("Enter your name: ")

# Reuse the variable
print(f"Hello, {name}!")

# Change the value
name = name.upper()

# Reuse again
print(f"Your name in uppercase: {name}")
```

Why do we need variable?

Variables in Python

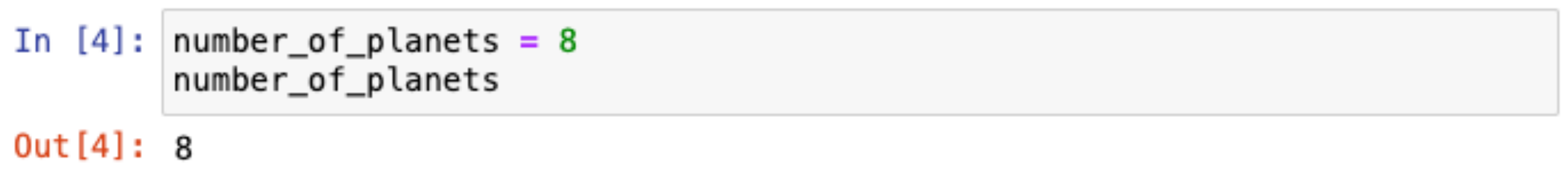
Assign a value to a variable via assignment operator “=”



```
In [1]: mystring = "It has been a lovely day."
        approx_pi = 3.1415
        number_of_planets = 9

In [2]: mystring

Out[2]: 'It has been a lovely day.'

In [3]: number_of_planets

Out[3]: 9
```

What are the types of my variables?

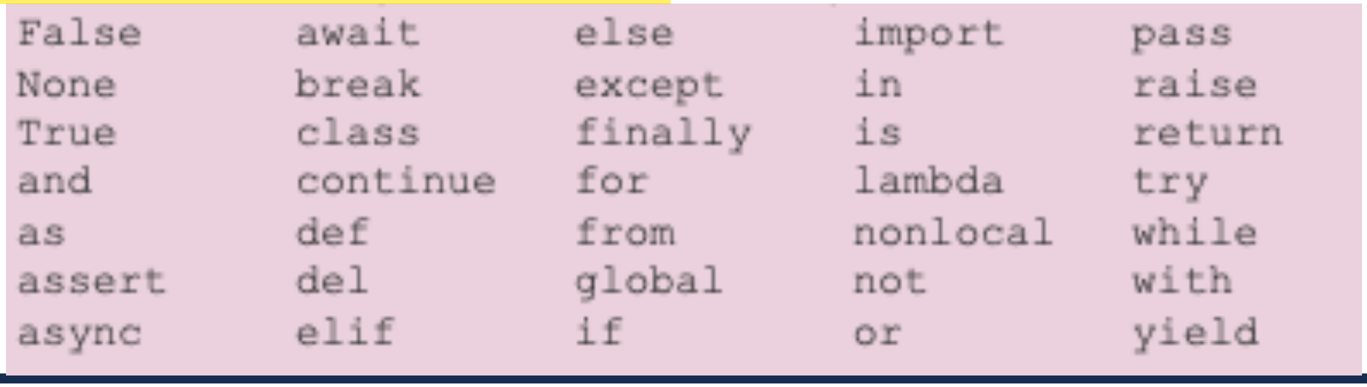
Change the value of a variable



```
In [4]: number_of_planets = 8
        number_of_planets

Out[4]: 8
```

Variable names in Python

Python variable names can be arbitrarily long, and may contain any letters, numbers and undercores(_), but may not start with a number. Variable can have any name, except for the python 3 reserved keywords



False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Key naming conventions for variables in Python:

1) Short but descriptive: **user_age** is preferable to **the_age_of_the_user** or **ua**.
2) Lowercase with underscores: **account_balance** or **number_of_planets**…
3) Constants, names of constants should be in all uppercase letters with underscores separating words. For example, **MAX_SIZE** or **DEFAULT_COLOR**

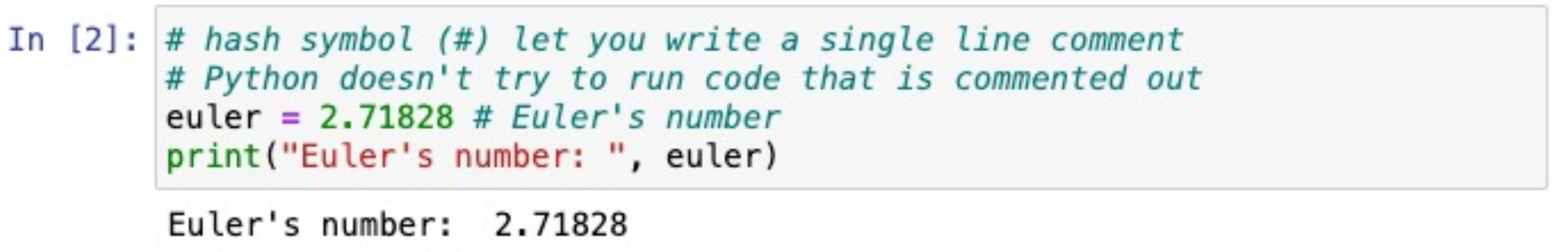
More can be found on: https://google.github.io/styleguide/pyguide.html

Talking about syntax -- comments

Comments provide a way to document your code.

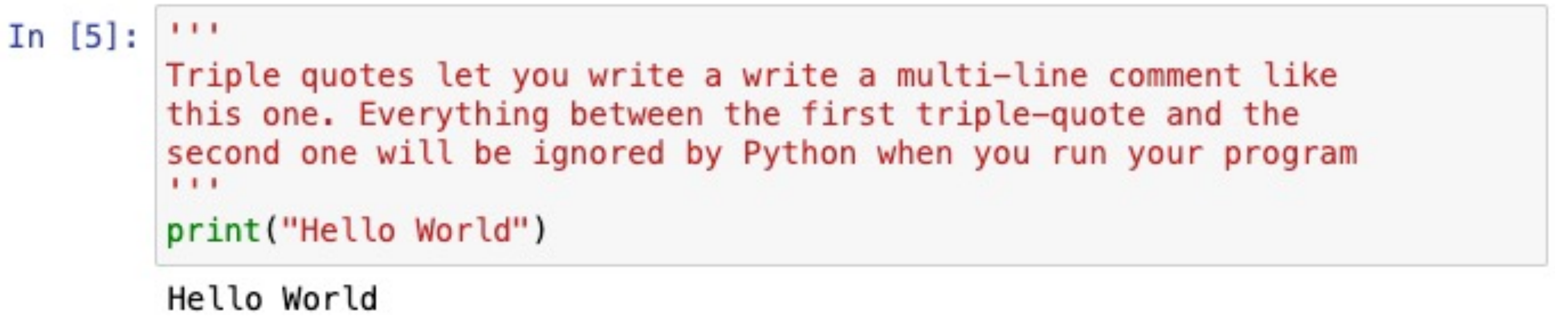
- **<u>Ignored</u>** by the Python interpreter during execution
- Notes and explanations within the code, more **<u>readable and maintainable</u>**

1) Single line comment



```
In [2]: # hash symbol (#) let you write a single line comment
        # Python doesn't try to run code that is commented out
        euler = 2.71828 # Euler's number
        print("Euler's number: ", euler)

        Euler's number:  2.71828
```

2) Multi-line comment



```
In [5]: '''
        Triple quotes let you write a write a multi-line comment like
        this one. Everything between the first triple-quote and the
        second one will be ignored by Python when you run your program
        '''
        print("Hello World")

        Hello World
```

Expressions

Combine objects and operators to form expressions.

- 4 + 3
- (507 * 12) / 3

Syntax for a simple expression:

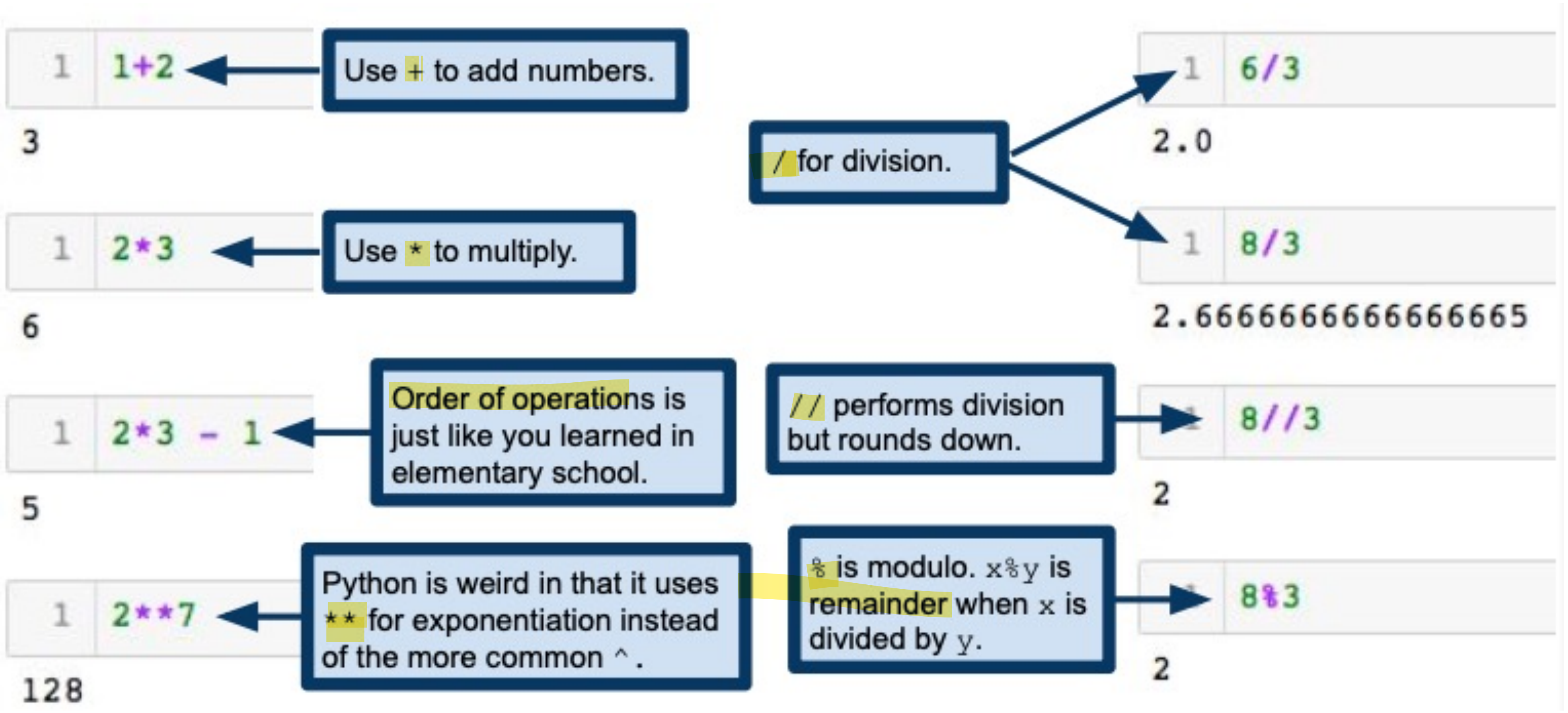
```
<object> <operator> <object>
```

Mathematical, Boolean and Conditional Expressions

Expression will always return a value with a **type**

Mathematical Expressions: int & float

There are **7** arithmetic operators in Python



```
1+2
3
```

Use + to add numbers.

```
2*3
6
```

Use * to multiply.

```
2*3 - 1
5
```

Order of operations is just like you learned in elementary school.

```
2**7
128
```

Python is weird in that it uses ** for exponentiation instead of the more common ^.

/ for division.

```
6/3
2.0
```

```
8/3
2.6666666666666665
```

// performs division but rounds down.

```
8//3
2
```

% is modulo. x%y is remainder when x is divided by y.

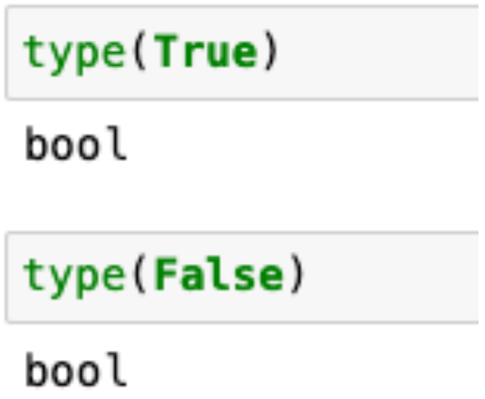
```
8%3
2
```

1. Data Types
2. Conditionals (very entry-level)
3. Functions

Boolean expressions

What: Boolean expressions evaluate the truth/falsity of a statement

Python supplies a special Boolean type, bool variable of type bool can either be True or False



```
type(True)
```

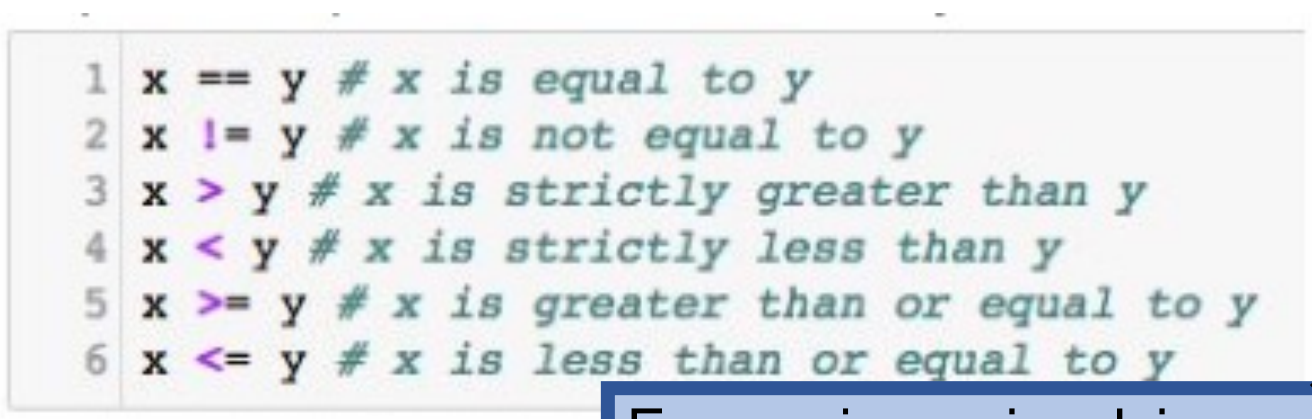
```
bool
```

```
type(False)
```

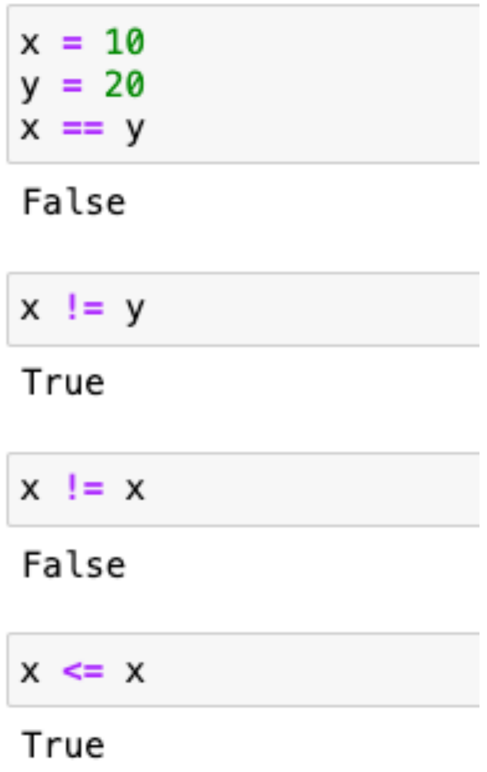
```
bool
```

Boolean expressions

Comparison operators available in Python:



```
x == y # x is equal to y
x != y # x is not equal to y
x > y # x is strictly greater than y
x < y # x is strictly less than y
x >= y # x is greater than or equal to y
x <= y # x is less than or equal to y
```



```
x = 10
y = 20
x == y
```

```
False
```

```
x != y
```

```
True
```

```
x != x
```

```
False
```

```
x <= x
```

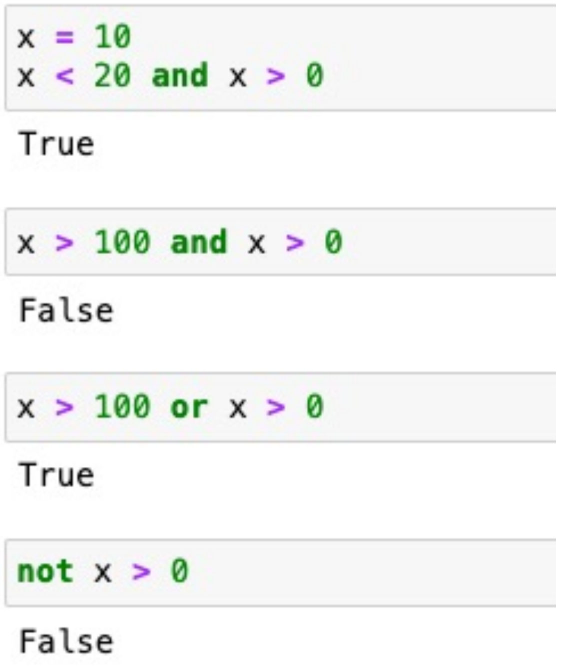
```
True
```

Expressions involving comparison operators evaluate to a Boolean

Note: in Pythonic style, one can compare many types, not just numbers. Most obviously, strings can be compared, with ordering given alphabetically.

Logical operators in Python

and, or and not



```
x = 10
x < 20 and x > 0
```

```
True
```

```
x > 100 and x > 0
```

```
False
```

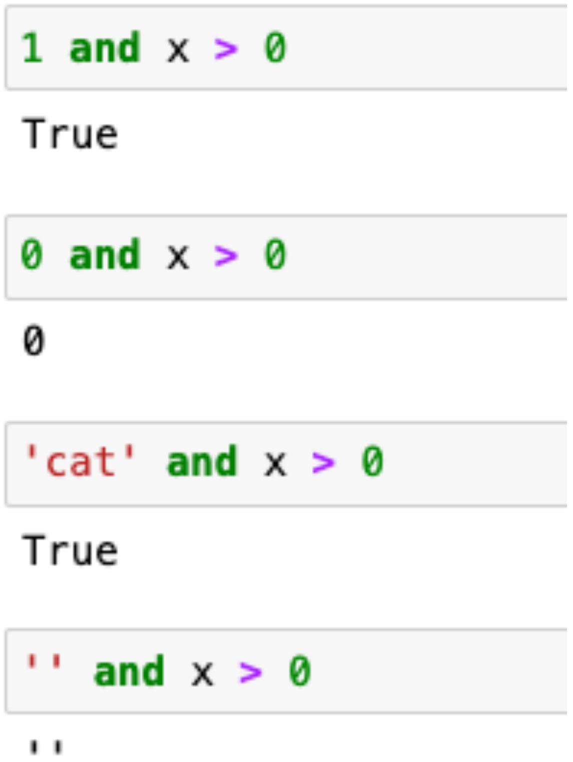
```
x > 100 or x > 0
```

```
True
```

```
not x > 0
```

```
False
```

Note: technically, any nonzero number or any nonempty string will evaluate to be True, but you should avoid comparing anything that isn't Boolean



```
1 and x > 0
```

```
True
```

```
0 and x > 0
```

```
0
```

```
'cat' and x > 0
```

```
True
```

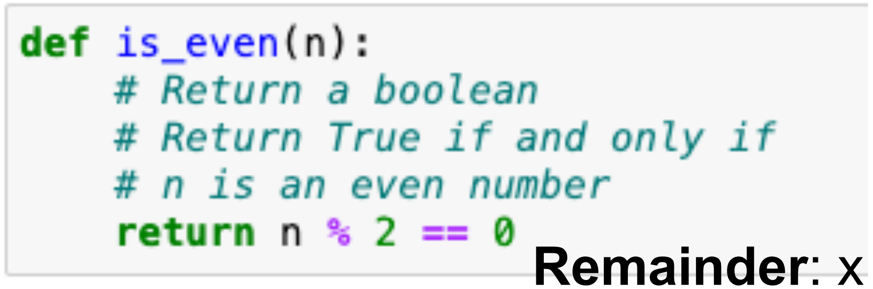
```
'' and x > 0
```

```
''
```

Python's and operator returns the first operand if it is false

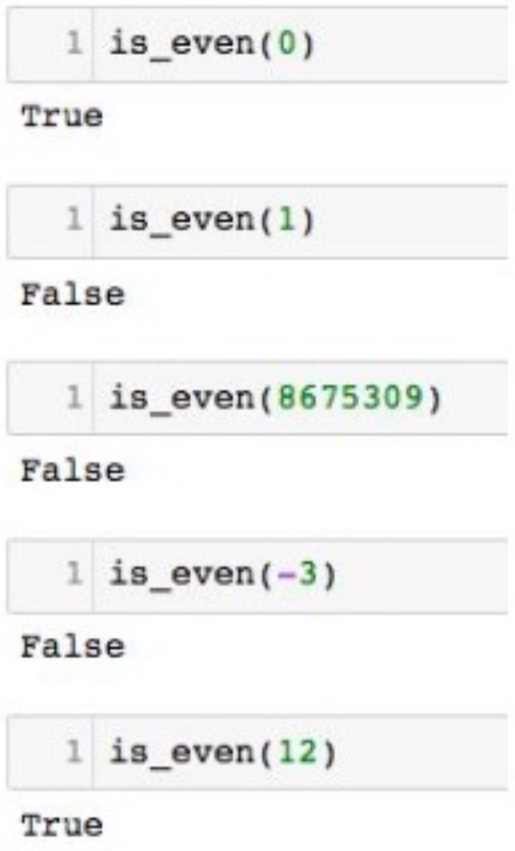
Boolean Expressions: Example

Let’s see Boolean expressions in action



```python
def is_even(n):
    # Return a boolean
    # Return True if and only if
    # n is an even number
    return n % 2 == 0
```

Remainder: x % y returns the remainder when x is divided by y.

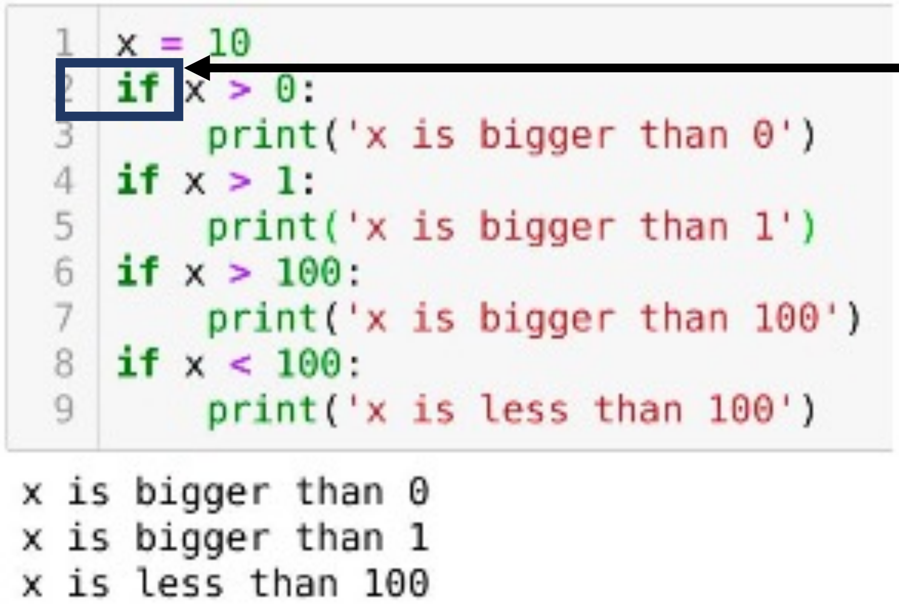


```
is_even(0)
True

is_even(1)
False

is_even(8675309)
False

is_even(-3)
False

is_even(12)
True
```

Note: in practice, we would want to include some extra code to check that n is actually a number, and to “fail gracefully” if it isn’t, e.g., by **throwing an error with a useful error message**. More about this in future lectures.

From Boolean to Conditional expressions

Sometimes we want to do different things depending on certain conditions



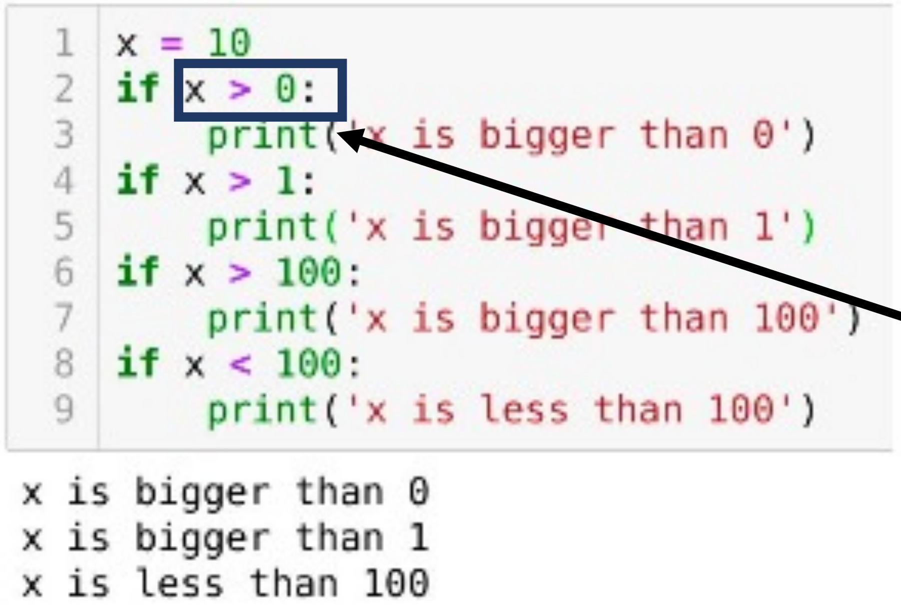
```python
x = 10
if x > 0:
    print('x is bigger than 0')
if x > 1:
    print('x is bigger than 1')
if x > 100:
    print('x is bigger than 100')
if x < 100:
    print('x is less than 100')
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

This is an **if-statement**.

From Boolean to Conditional expressions

Sometimes we want to do different things depending on certain conditions



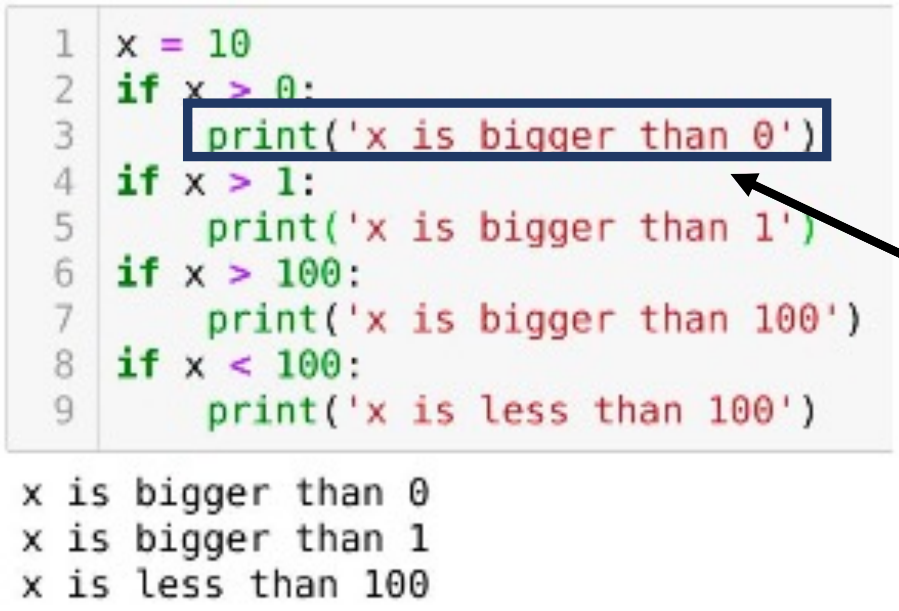
```
x = 10
if x > 0:
    print('x is bigger than 0')
if x > 1:
    print('x is bigger than 1')
if x > 100:
    print('x is bigger than 100')
if x < 100:
    print('x is less than 100')
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

This Boolean expression is called the test condition, or just the condition.

From Boolean to Conditional expressions

Sometimes we want to do different things depending on certain conditions



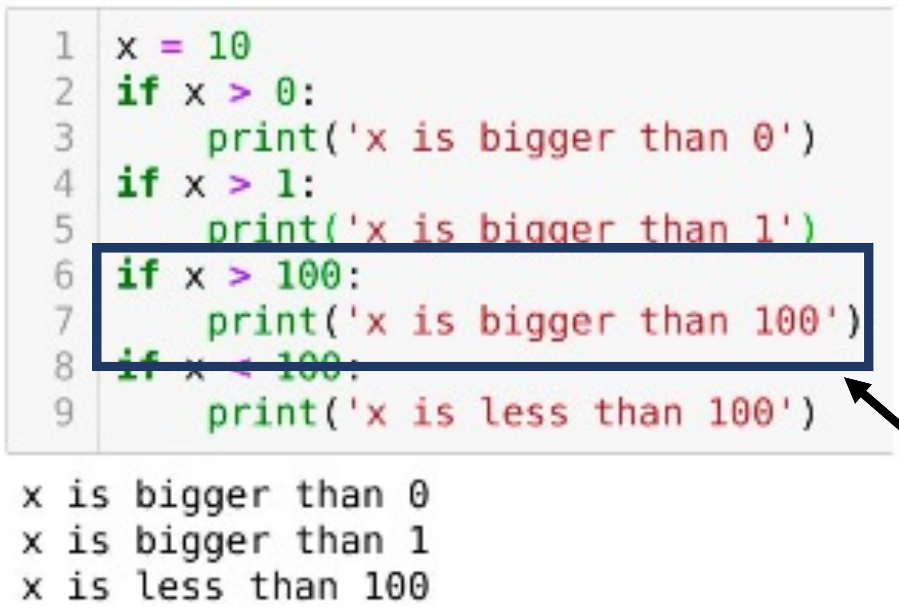
```
x = 10
if x > 0:
    print('x is bigger than 0')
if x > 1:
    print('x is bigger than 1')
if x > 100:
    print('x is bigger than 100')
if x < 100:
    print('x is less than 100')
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

If the condition evaluates to the True, then Python runs the code in the body of the if-statement.

From Boolean to Conditional expressions

Sometimes we want to do different things depending on certain conditions



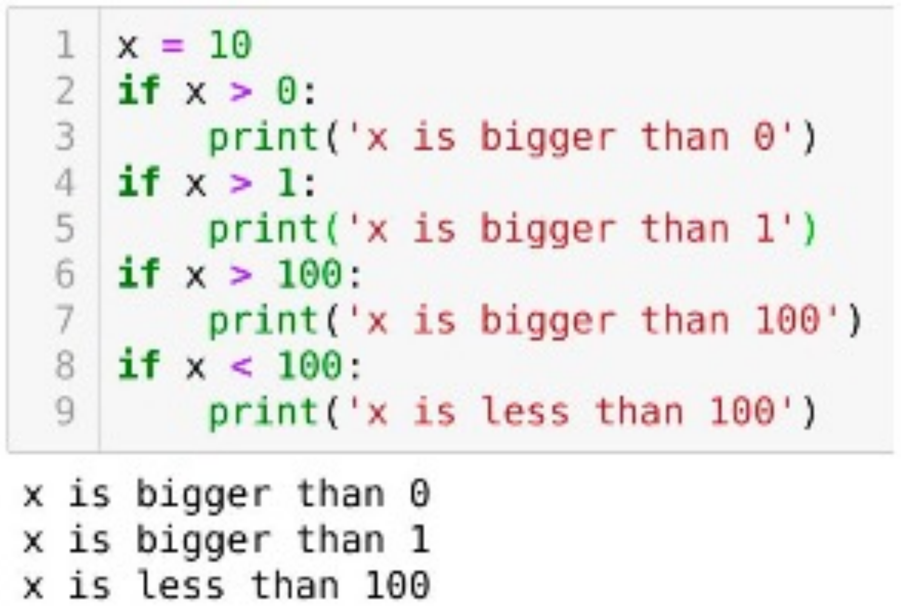
```
x = 10
if x > 0:
    print('x is bigger than 0')
if x > 1:
    print('x is bigger than 1')
if x > 100:
    print('x is bigger than 100')
if x < 100:
    print('x is less than 100')
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

If the condition evaluates to False, then Python skips the body and continues running code starting at the end of the if-statement

From Boolean to Conditional expressions

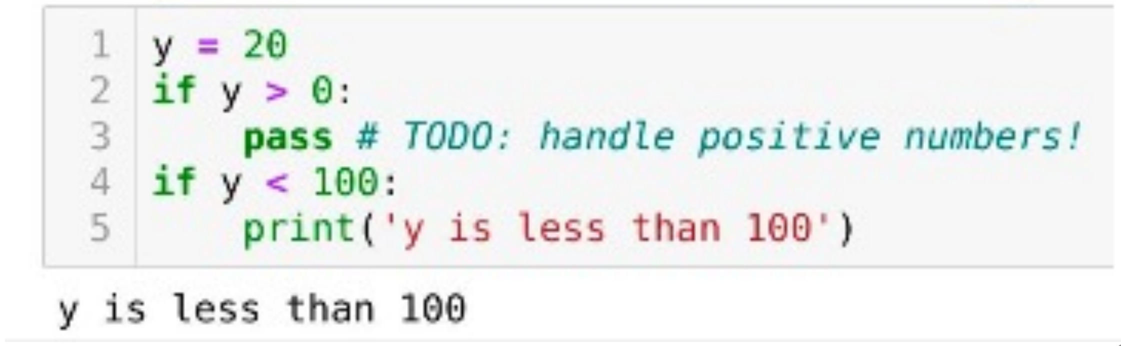
Sometimes we want to do different things depending on certain conditions



```python
x = 10
if x > 0:
    print('x is bigger than 0')
if x > 1:
    print('x is bigger than 1')
if x > 100:
    print('x is bigger than 100')
if x < 100:
    print('x is less than 100')
```

```
x is bigger than 0
x is bigger than 1
x is less than 100
```

Note: the body of a conditional statement can have any number of lines in it, but it must have at least one line. To do nothing, use the pass keyword.

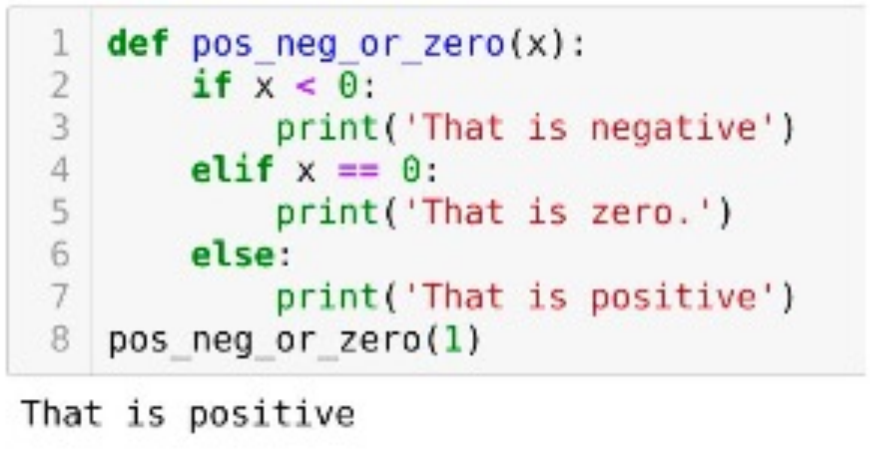


```python
y = 20
if y > 0:
    pass # TODO: handle positive numbers!
if y < 100:
    print('y is less than 100')
```

```
y is less than 100
```

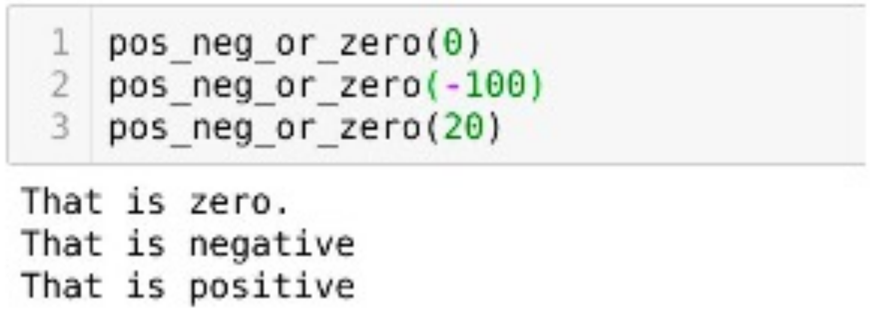
Chained conditionals

More complicated logic can be handled with chained conditionals



```python
def pos_neg_or_zero(x):
    if x < 0:
        print('That is negative')
    elif x == 0:
        print('That is zero.')
    else:
        print('That is positive')
pos_neg_or_zero(1)
```

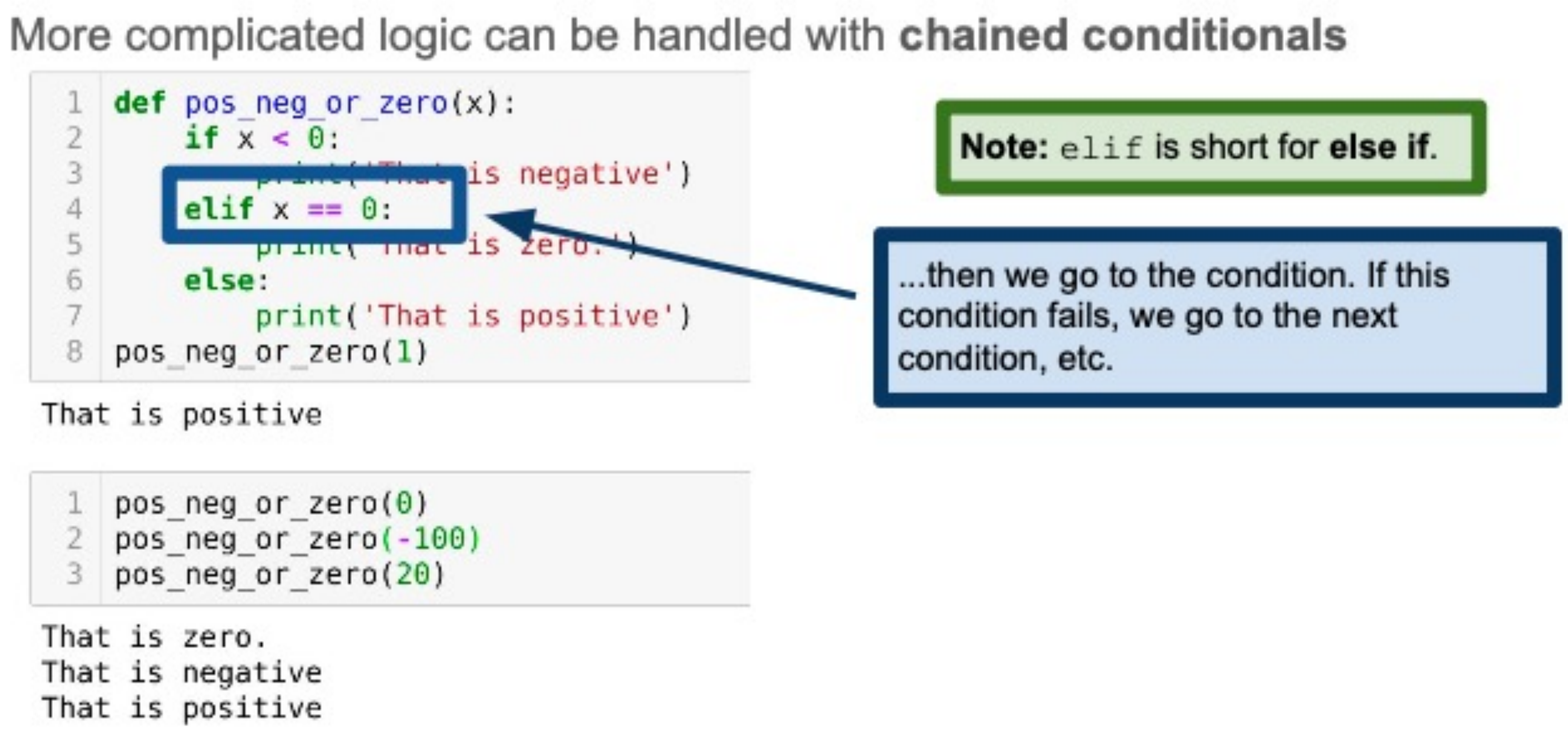
```
That is positive
```



```python
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Chained conditionals



More complicated logic can be handled with **chained conditionals**

```python
def pos_neg_or_zero(x):
    if x < 0:
        print('That is negative')
    elif x == 0:
        print('That is zero.')
    else:
        print('That is positive')
pos_neg_or_zero(1)
```

```
That is positive
```

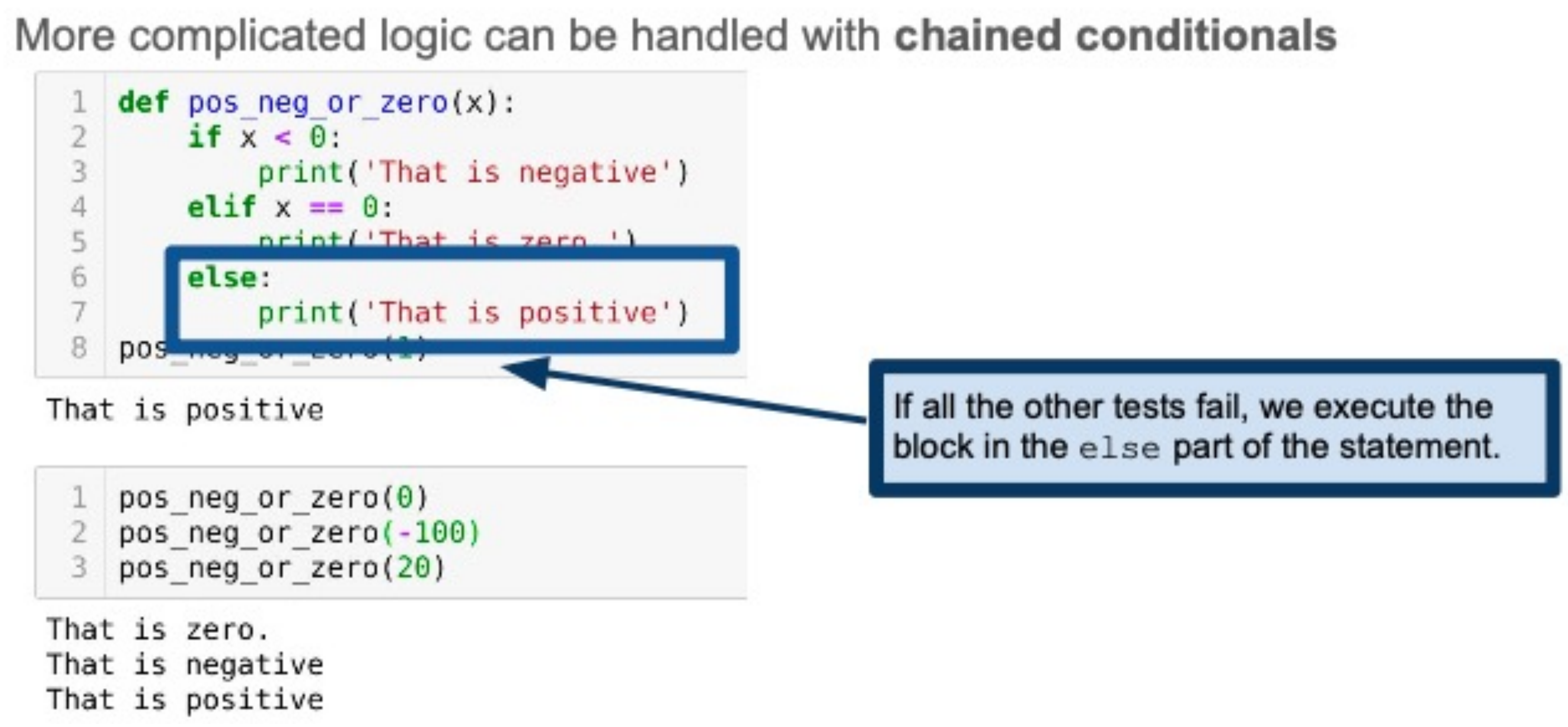
```python
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Note: `elif` is short for **else if**.

...then we go to the condition. If this condition fails, we go to the next condition, etc.

Chained conditionals



More complicated logic can be handled with **chained conditionals**

```
def pos_neg_or_zero(x):
    if x < 0:
        print('That is negative')
    elif x == 0:
        print('That is zero.')
    else:
        print('That is positive')
pos_neg_or_zero(1)
```

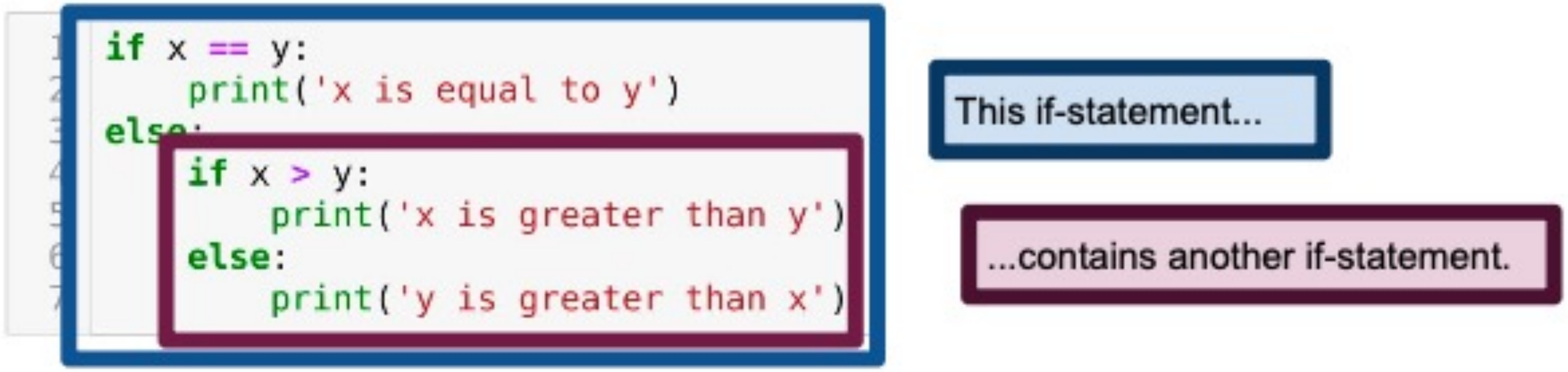
```
That is positive
```

```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

If all the other tests fail, we execute the block in the `else` part of the statement.

Conditionals can be nested



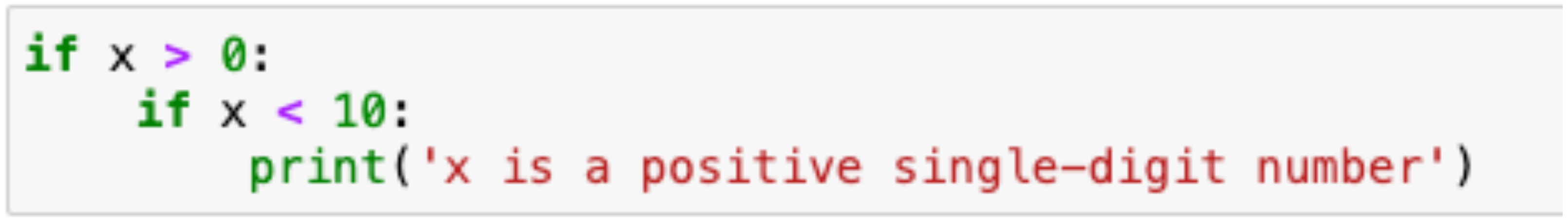
```
if x == y:
    print('x is equal to y')
else:
    if x > y:
        print('x is greater than y')
    else:
        print('y is greater than x')
```

Nested condition can be simplified

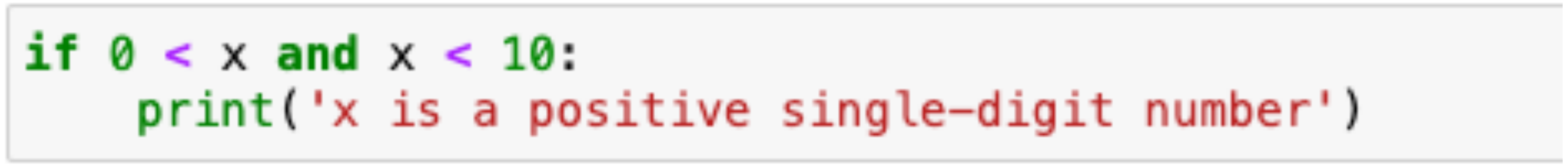
Often, a nested condition can be simplified.

When possible, I recommend for the sake of your sanity,

because debugging complicated nested conditions is tricky!



```python
if x > 0:
    if x < 10:
        print('x is a positive single-digit number')
```



```python
if 0 < x and x < 10:
    print('x is a positive single-digit number')
```

Those two if-statement are equivalent, in that they do the same thing, but second is (arguably) preferable, as it is simpler to read

1. Data Types
2. Conditionals
3. Functions

Intro: Functions in Python

We've already seen examples of functions: e.g.,

type(), print(), int()…

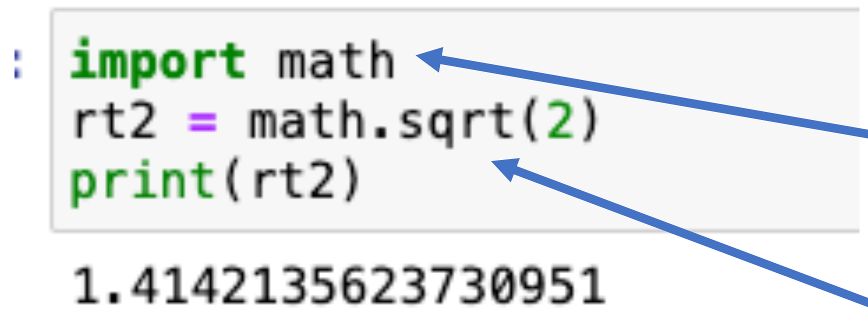
Function calls take the form:

function_name(function arguments)

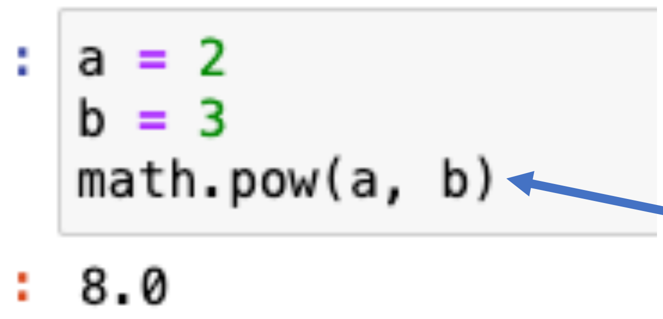
A function takes zero or more **arguments** and **returns** a value

Calling functions in Python

A function takes zero or more **arguments** and **returns** a value
The value can be any type…



```
import math
rt2 = math.sqrt(2)
print(rt2)

1.4142135623730951
```



```
a = 2
b = 3
math.pow(a, b)

8.0
```

Python math **module** provides a number of math functions. We have to **import** (i.e., load) the module before we can use it.

math.sqrt() takes one argument, returns its square root.

math.pow() takes two arguments. Returns the value obtained by raising the first to the power of the second

Modules in Python

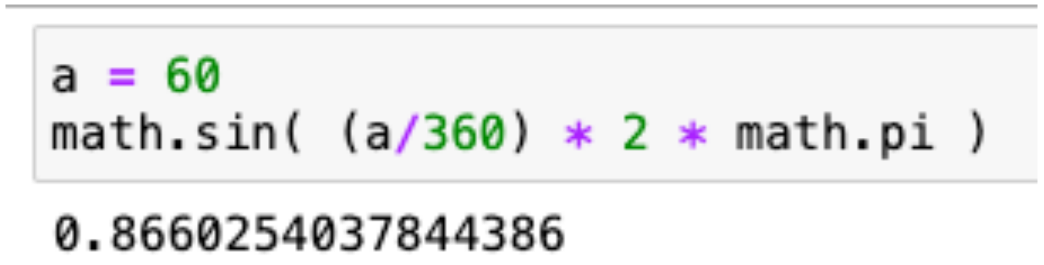
A file containing Python definitions, **functions** and statements. A module can define variables, functions, classes (which we will cover later), as well as runnable code.

Documentation for the Python modules:
https://docs.python.org/3/tutorial/modules.html

Documentation for the Python math module:
https://docs.python.org/3/library/math.html

Functions can be composed

Supply an expression as the argument of a function

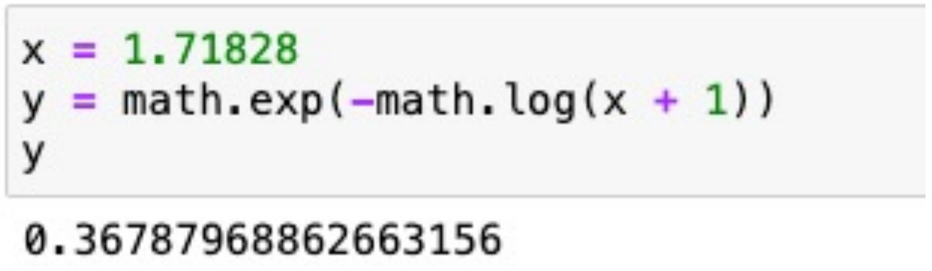


```
a = 60
math.sin( (a/360) * 2 * math.pi )
```

```
0.8660254037844386
```

math.sin() has an expression as its argument, which has to be evaluated before we can compute the answer.

Output of one function becomes input to another



```
x = 1.71828
y = math.exp(-math.log(x + 1))
y
```

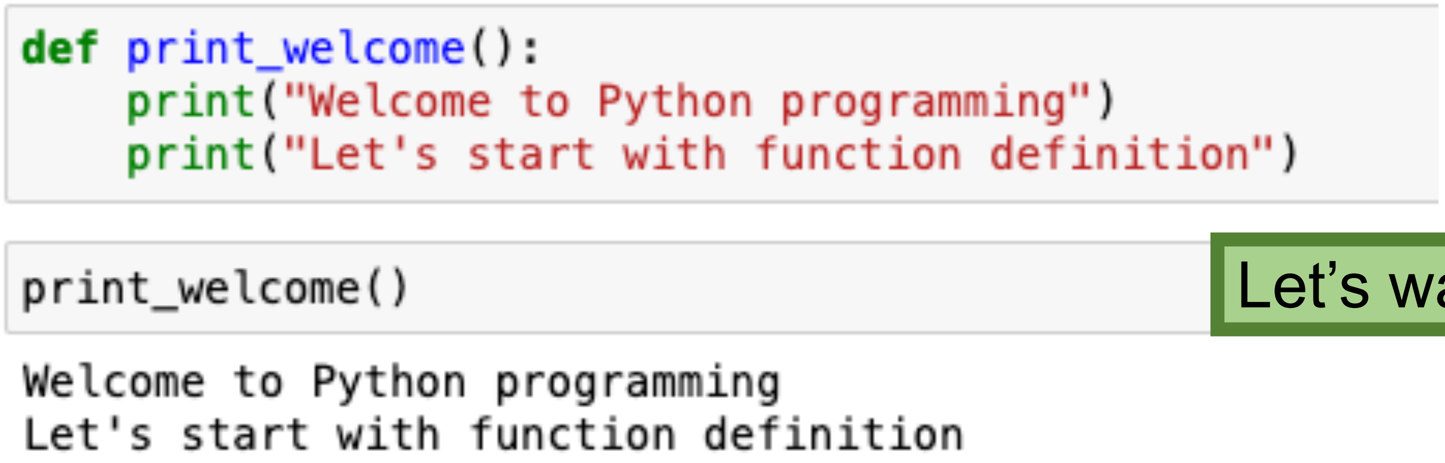
```
0.3678796862663156
```

Functions can even have the output of another function as argumments

Make new functions (define functions)

All about making/writing new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_welcome():
    print("Welcome to Python programming")
    print("Let's start with function definition")
```

```
print_welcome()
```

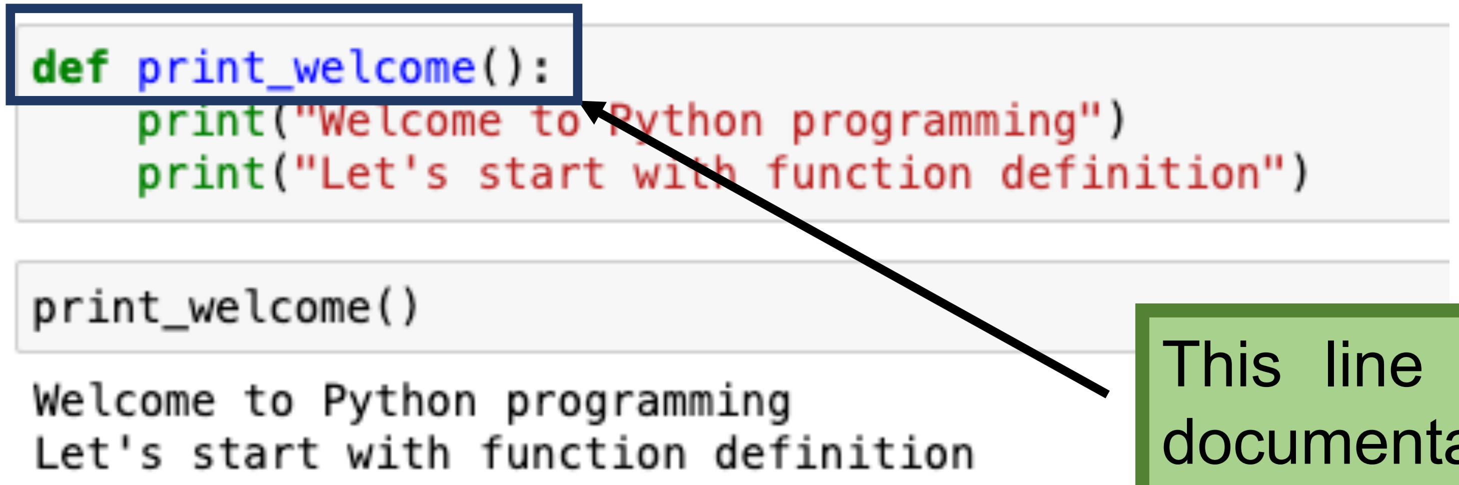
```
Welcome to Python programming
Let's start with function definition
```

Let’s walk through this line by line.

Defining functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_welcome():
    print("Welcome to Python programming")
    print("Let's start with function definition")
```

```
print_welcome()
```

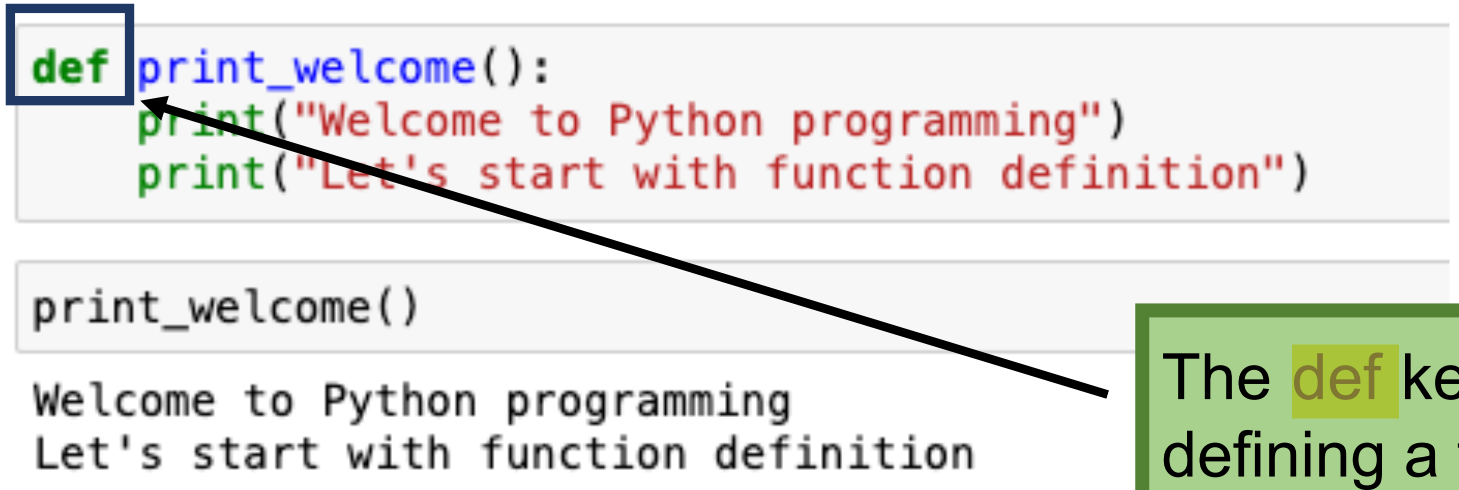
```
Welcome to Python programming
Let's start with function definition
```

This line (called the **header** in some documentation) says that we are defining a function called print_welcome, and that the function takes no argument

Defining functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_welcome():
    print("Welcome to Python programming")
    print("Let's start with function definition")
```

```
print_welcome()
```

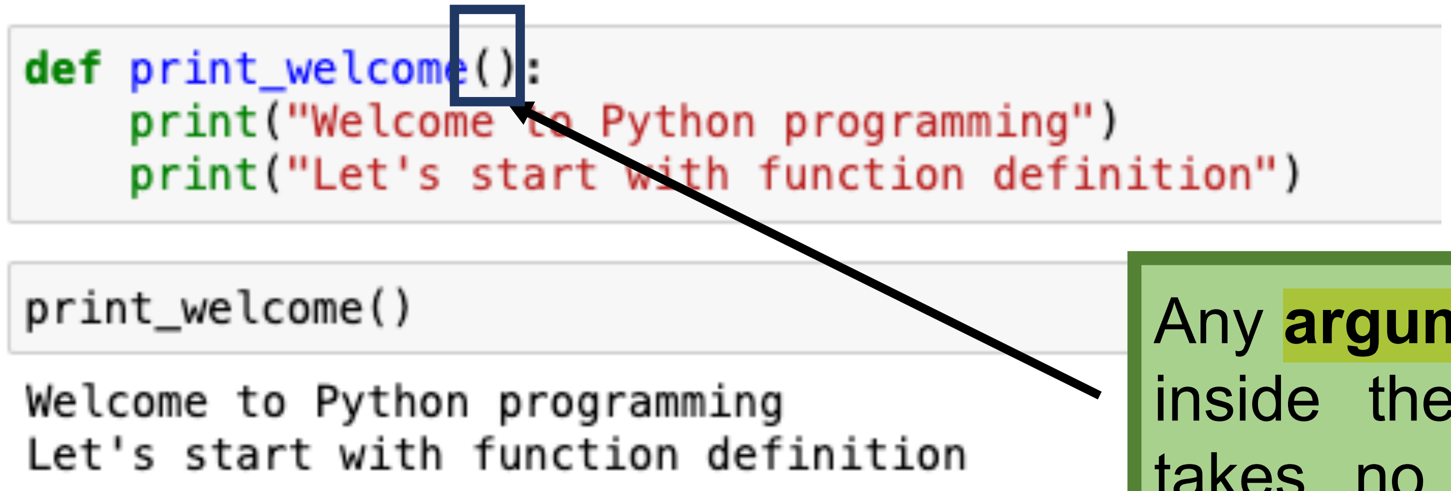
```
Welcome to Python programming
Let's start with function definition
```

The def keyword tells python that we are defining a function.

Defining functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_welcome():
    print("Welcome to Python programming")
    print("Let's start with function definition")
```

```
print_welcome()
```

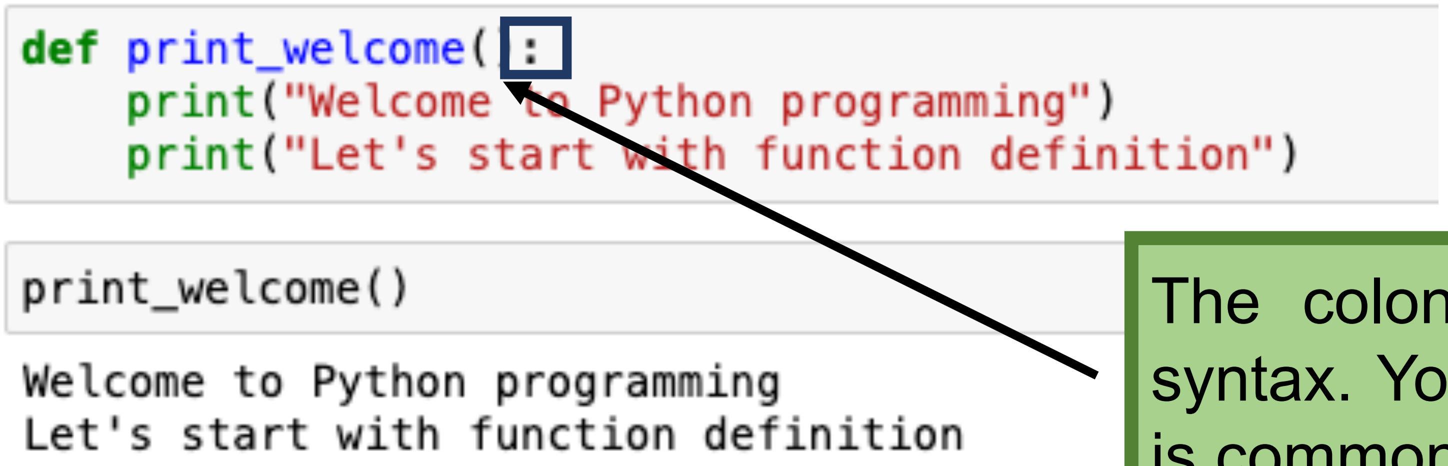
```
Welcome to Python programming
Let's start with function definition
```

Any **arguments** to the function are given inside the **parenthesis**. This function takes no argument, so we just give empty parenthesis. In a few slides, we'll see a function that takes arguments

Defining functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_welcome():
    print("Welcome to Python programming")
    print("Let's start with function definition")
```

```
print_welcome()
```

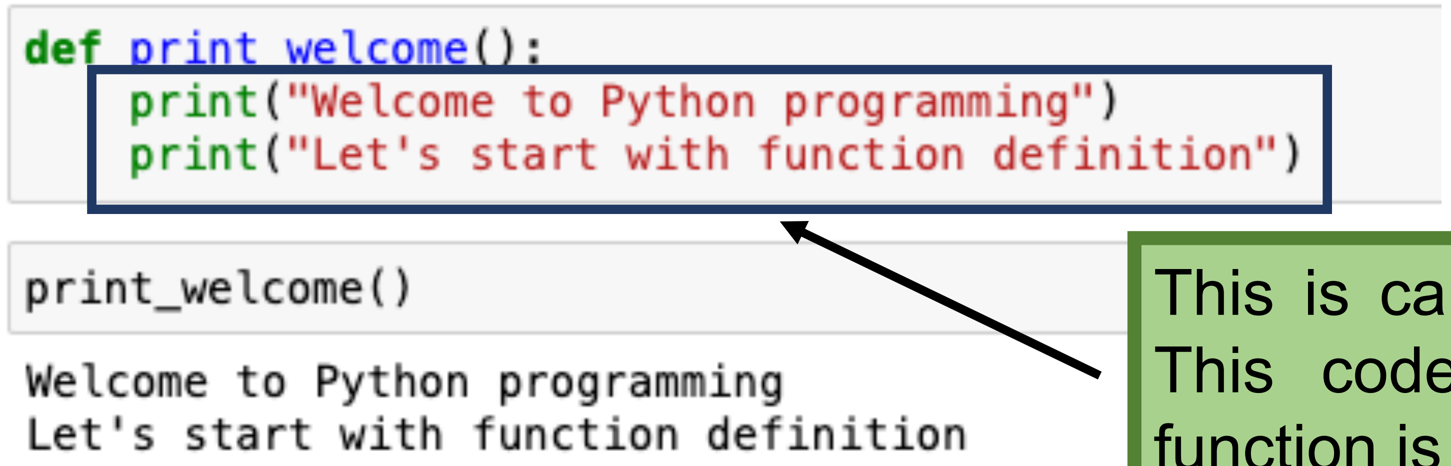
```
Welcome to Python programming
Let's start with function definition
```

The colon (:) is required by Python's syntax. You'll see this symbol a lot, as it is commonly used in Python to <u>signal the start of an indented block of code</u>. (More on this in a few slides).

Defining functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_welcome():
    print("Welcome to Python programming")
    print("Let's start with function definition")
```

```
print_welcome()
```

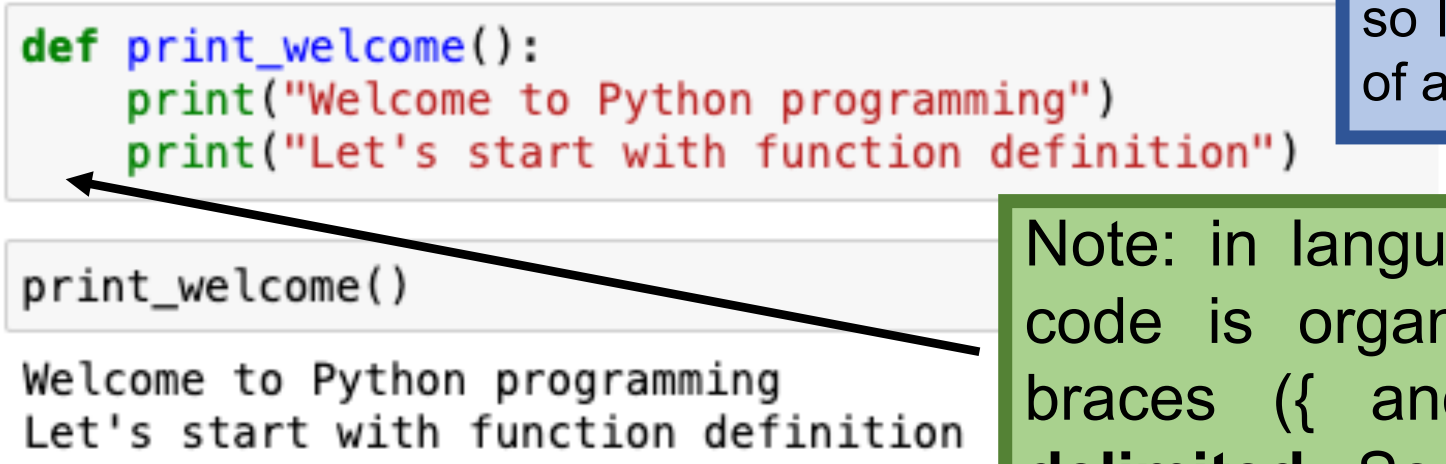
```
Welcome to Python programming
Let's start with function definition
```

This is called the **body** of the function. This code is executed whenever the function is called.

Defining functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```python
def print_welcome():
    print("Welcome to Python programming")
    print("Let's start with function definition")
```

```python
print_welcome()
```

```
Welcome to Python programming
Let's start with function definition
```

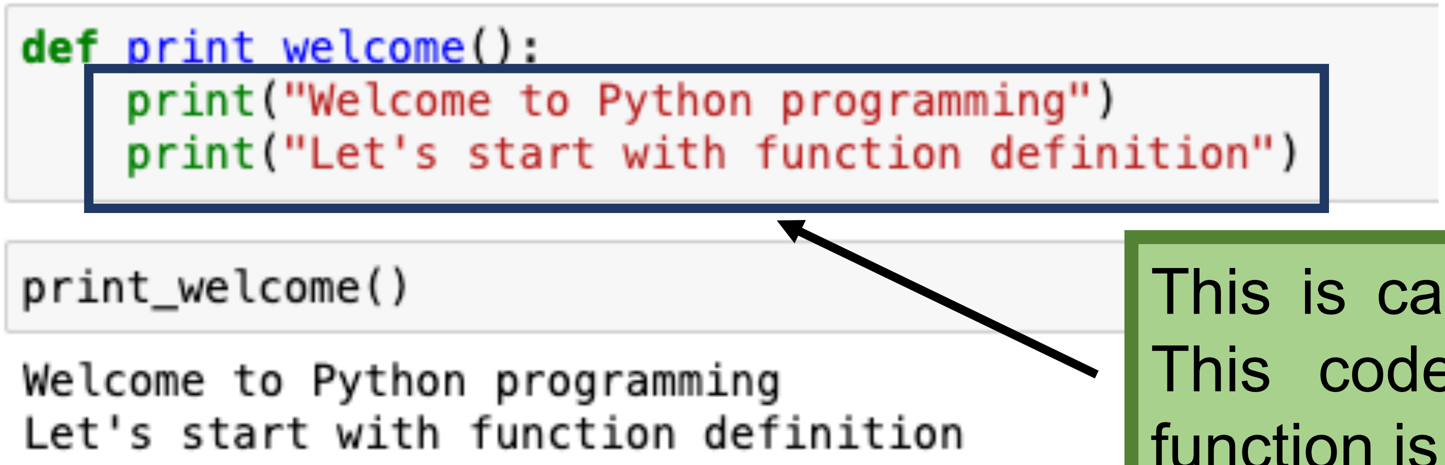
This whitespace can be tabs, or spaces, so long s it's consistent. It is taken care of automatically by most IDEs.

Note: in languages like R, C/C++ and Java, code is organized into **blocks** using curly braces ({ and }). Python is **whitespace delimited**. So we tell Python which lines of code are part of the function definition using **indentation**.

Defining functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_welcome():
    print("Welcome to Python programming")
    print("Let's start with function definition")
```

```
print_welcome()
```

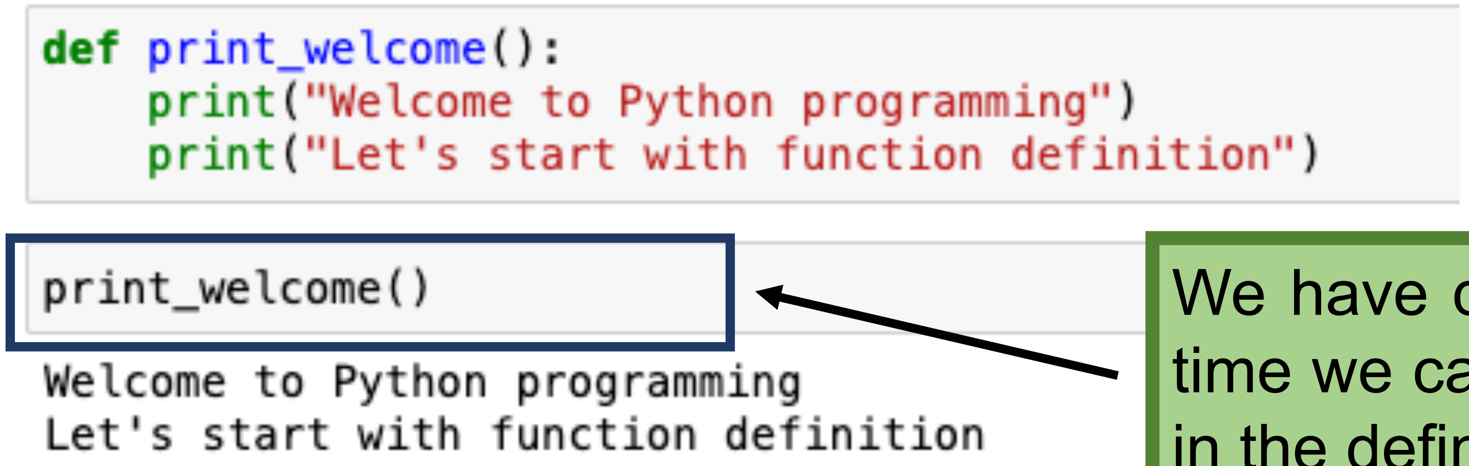
```
Welcome to Python programming
Let's start with function definition
```

This is called the **body** of the function. This code is executed whenever the function is called.

Defining functions

We can make new functions using **function definition**

Creates a new function, which we can then call whenever we need it



```
def print_welcome():
    print("Welcome to Python programming")
    print("Let's start with function definition")
```

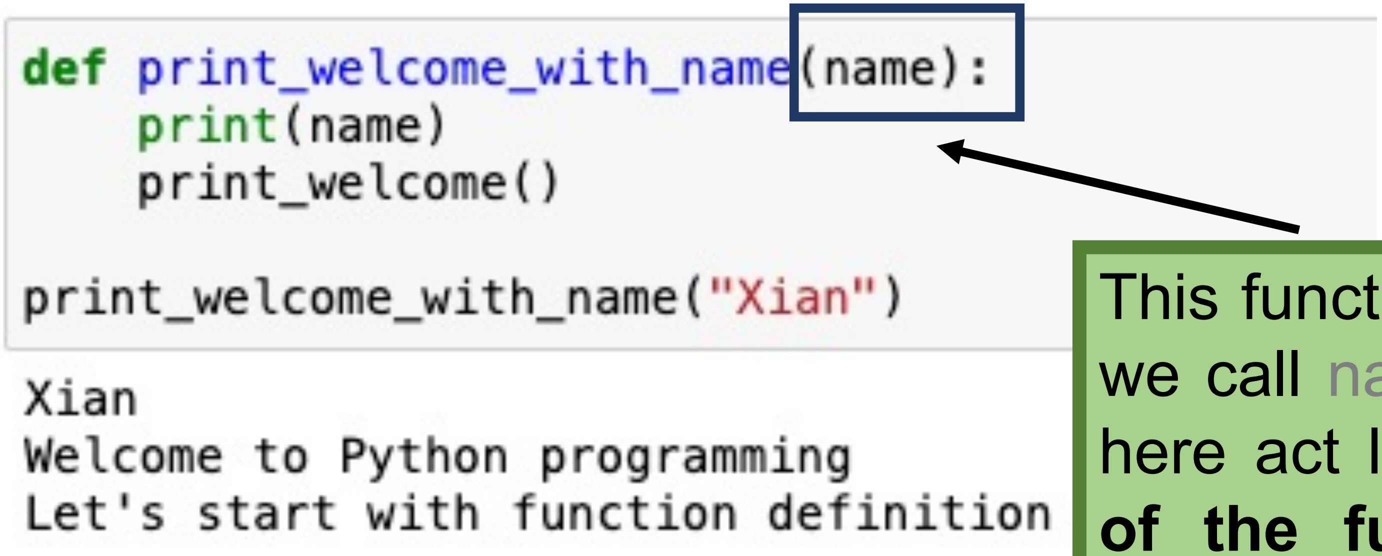
```
print_welcome()
```

```
Welcome to Python programming
Let's start with function definition
```

We have defined our function. Now, any time we call it, Python executes the code in the definition, in order.

Defining functions

After defining a function, we can use it anywhere, including in other functions



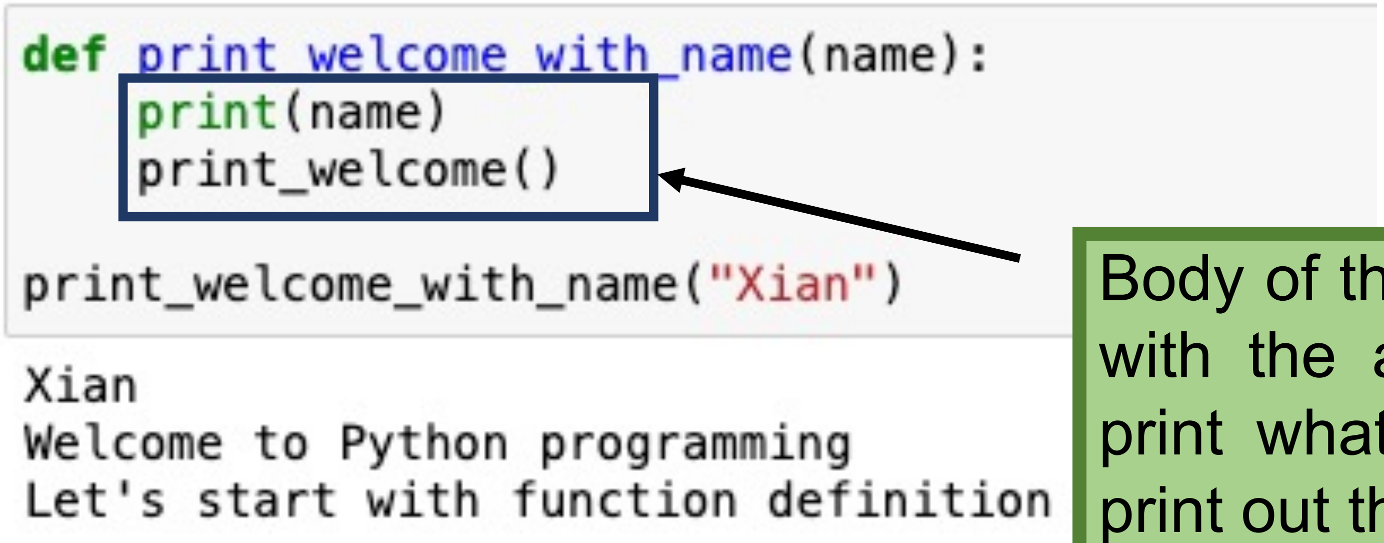
```
def print_welcome_with_name(name):
    print(name)
    print_welcome()

print_welcome_with_name("Xian")
```

```
Xian
Welcome to Python programming
Let's start with function definition
```

This function takes one argument, which we call name. All the arguments named here act like variables **within the body of the function**, but not outside the body. We'll return to this in a few slides.

Defining functions

After defining a function, we can use it anywhere, including in other functions



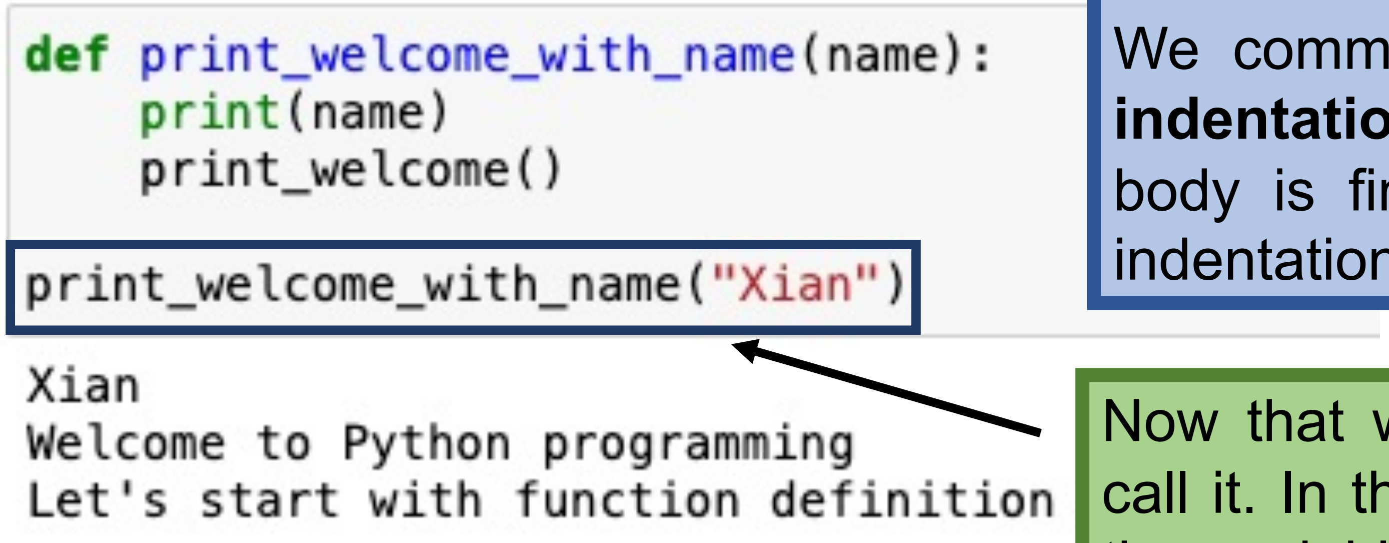
```
def print_welcome_with_name(name):
    print(name)
    print_welcome()

print_welcome_with_name("Xian")
```

```
Xian
Welcome to Python programming
Let's start with function definition
```

Body of the function specifies what to do with the argument(s). In this case, we print whatever the argument was, then print out the message in print_welcome.

Defining functions

After defining a function, we can use it anywhere, including in other functions



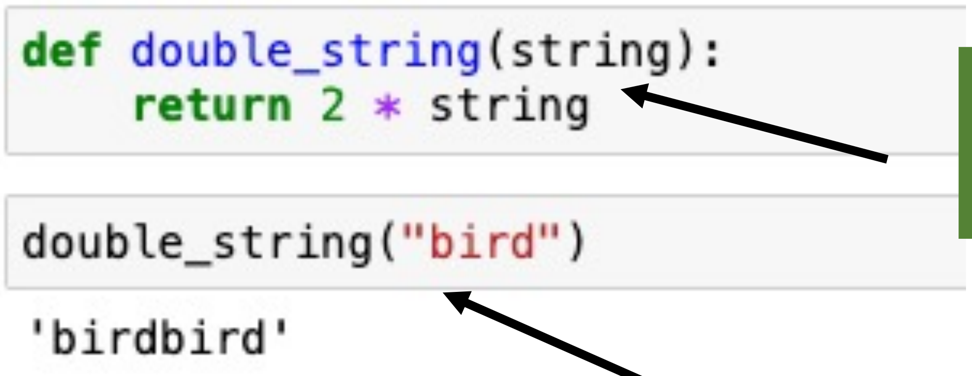
```
def print_welcome_with_name(name):
    print(name)
    print_welcome()

print_welcome_with_name("Xian")

Xian
Welcome to Python programming
Let's start with function definition
```

Note: this last line is not part of the function. We communicate this fact to Python by the **indentation**. Python knows that the function body is finished once it sees a line without indentation.

Now that we've defined the function, we can call it. In this case, when we call our function, the variable name in the definition gets the value "xian", and then proceeds to run the code in the function body

The return keyword

Using the `return` keyword, we can define functions that produce results.



```
def double_string(string):
    return 2 * string
```

```
double_string("bird")
```

```
'birdbird'
```

double_string takes one argument, and returns that string, concatenated with itself

So when Python executes this line, it takes the string ‘bird’, which becomes the parameter string in the function double_string , and this line evaluates to the string ‘birdbird’.

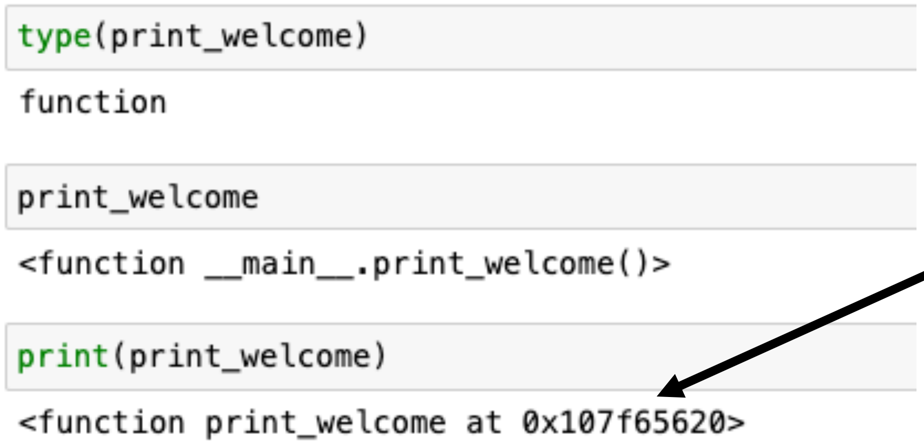
In class practice

One last thing…

Function is a type:

When you define a function, you are actually creating a variable of type **function**

Functions are objects that you can treat just like other variables.



```
type(print_welcome)
```

```
function
```

```
print_welcome
```

```
<function __main__.print_welcome()>
```

```
print(print_welcome)
```

```
<function print_welcome at 0x107f65620>
```

This number is the address in memory where print_welcome is stored. It may be different in your computer.

Things to do:

Install Anaconda and Jupyter

Familiarize yourself with Jupyter:
https://docs.jupyter.org/en/latest/start/index.html

Sign up for github and create your own stats 507 repo.

Read ch1-6 in Python 4 Everybody

Other things

HW1 posted on canvas.

If you run into trouble, attend GSI office hours for help.

- You can also post to the canvas discussion board.
- If you're having trouble, at least one of your classmates is, too.
- You'll learn more by explaining things to teach other than by reading posts.

Coming next:

Iteration, Strings, Lists, and Dictionaries.