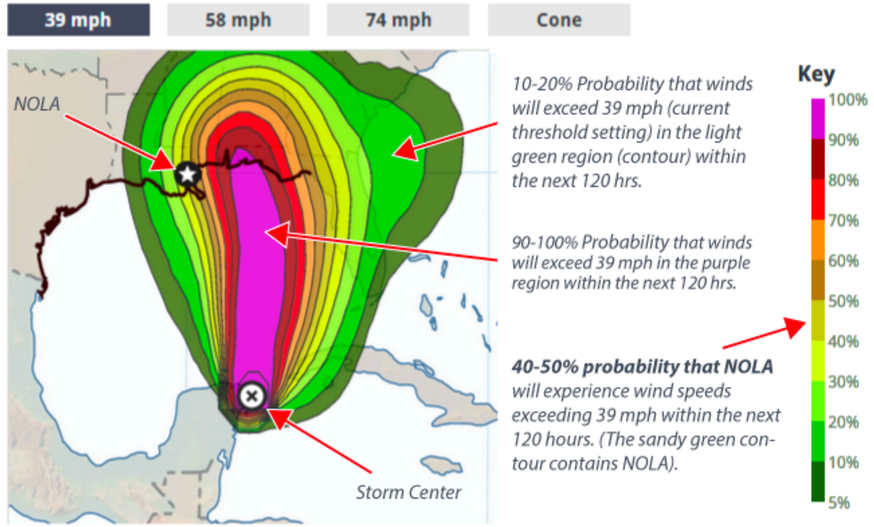
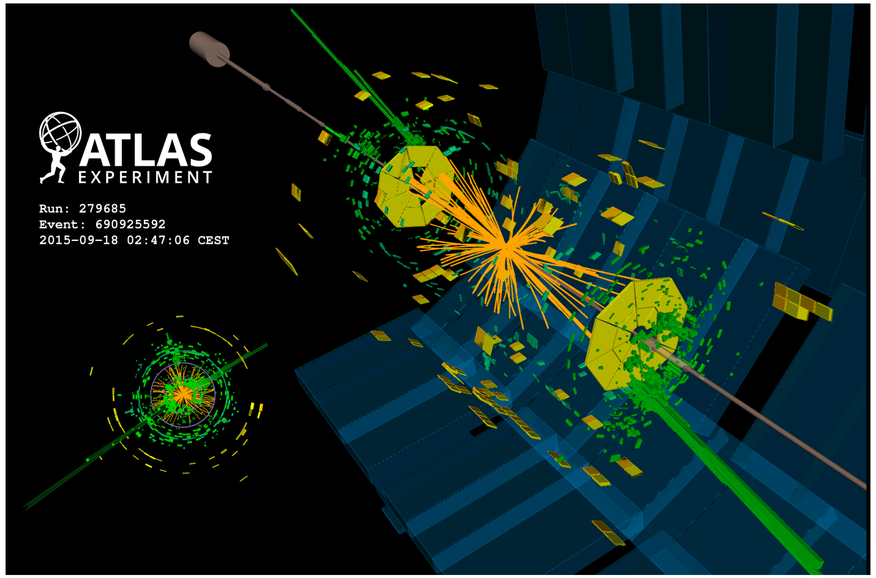
Python Programming

Kevin Nelson, Jianming Qian, Alexander Takla
Michigan Math and Science Scholars
24 July 2023

Why programming?

- **Physicists often *need* to use computer programs for several reasons:**
  - There is too much data to analyze by hand
  - In a computer program, you can simulate many possibilities very quickly:





1 of 10 billion "events" in ATLAS

Simulate 1 hurricane many times
Changing parameters

- Some physical systems do not have neat solutions written in terms of simple variables like x, x^2 but are described by a complicated connected system of differential equations

Kevin Nelson

What is a computer?

- First and foremost a computer is DUMB
- The computer is made up of an enormous number of transistors, which store binary (0 or 1) information
- The computer can execute *algorithms*, or a set of instructions
- The computer cannot think for itself!!

What is an algorithm?

- **An algorithm is a set of step by step instructions for completing a task**
- **We use algorithms every day:**
  - Brushing your teeth: 1. Put toothpaste on brush. 2. Spend 2 minutes brushing. 3. Rinse mouth.
  - Following a recipe: 1. Mix ingredients. 2. Bake at 350 degrees F for 15 minutes. 3. Let cool.
  - If you followed these algorithms in reverse order, they make no sense! They would not get the job done.
- **A computer is a device which can follow algorithms.**
- **But, the computer needs the algorithm to be expressed in a language it can understand: a programming language.**

What is a programming language?

- **A programming language has its own rules of grammar and syntax, just like any other language (English, Spanish, etc.).**

- **Examples of common programming languages:**
  - **Python**, C, C++, C#, Java, Javascript, HTML, LaTeX, MATLAB, PHP, SQL… there are thousands, even silly ones

```
>>> print("Hello World!")
Hello World!
```

Some programming languages are more useful than others

Choice of language depends on the problem at hand

Examples

Hello, world!

```
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken
chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken chicken chicken chicken chicken
chicken chicken chicken chicken chicken chicken
```

How does a computer read the language?

- **Ultimately, the computer “speaks” only binary. 1s and 0s.**

- **After we write python code, it is translated to a “lower level” language that the computer can read**

```
>>> print("Hello World!")
Hello World!
```

```
org 0x100


mov dx, msg
mov ah, 9
int 0x21

mov dl, 0x0d
mov ah, 2
int 0x21

mov dl, 0x0a
mov ah, 2
int 0x21

mov ah, 0x4c
int 0x21
```

Outline

- **The programming language we will use in this course is Python**
  - Easy to learn, but difficult to master. Python hides many details, but then learning them gets harder later.
  - We are quite literally learning a new language! It is difficult and takes time. In the two weeks here you will get a very abbreviated crash course, but it takes *years* to become an adept programmer.
- **Outline:**
  - Variables, values, and expressions
  - Lists and strings
  - Loops and repetition
  - Choice and “Control Flow”
  - Functions and code reuse
  - Errors
  - Libraries
  - Jupyter Notebooks

Variables, Values, Expressions

Variables

Any Python interpreter can be used as a calculator:

PYTHON

```
3 + 5 * 4
```

OUTPUT

```
23
```

This is great but not very interesting. To do anything useful with data, we need to assign its value to a *variable*. In Python, we can assign a value to a variable, using the equals sign `=`. For example, we can track the weight of a patient who weighs 60 kilograms by assigning the value `60` to a variable `weight_kg`:

PYTHON

```
weight_kg = 60
```

From now on, whenever we use `weight_kg`, Python will substitute the value we assigned to it. In layperson's terms, **a variable is a name for a value**.

In Python, variable names:

- can include letters, digits, and underscores
- cannot start with a digit
- are case sensitive.

This means that, for example:

- `weight0` is a valid variable name, whereas `0weight` is not
- `weight` and `Weight` are different variables

Types of data

Python knows various types of data. Three common ones are:

- integer numbers
- floating point numbers, and
- strings.

In the example above, variable `weight_kg` has an integer value of `60`. If we want to more precisely track the weight of our patient, we can use a floating point value by executing:

PYTHON

```
weight_kg = 60.3
```

To create a string, we add single or double quotes around some text. To identify and track a patient throughout our study, we can assign each person a unique identifier by storing it in a string:

PYTHON

```
patient_id = '001'
```

Using Variables in Python

Once we have data stored with variable names, we can make use of it in calculations. We may want to store our patient's weight in pounds as well as kilograms:

PYTHON

```
weight_lb = 2.2 * weight_kg
```

We might decide to add a prefix to our patient identifier:

PYTHON

```
patient_id = 'inflam_' + patient_id
```

Built-in Python functions

To carry out common tasks with data and variables in Python, the language provides us with several built-in functions. To display information to the screen, we use the `print` function:

PYTHON

```
print(weight_lb)
print(patient_id)
```

OUTPUT

```
132.66
inflam_001
```

When we want to make use of a function, referred to as calling the function, we follow its name by parentheses. The parentheses are important: if you leave them off, the function doesn't actually run! Sometimes you will include values or variables inside the parentheses for the function to use. In the case of `print`, we use the parentheses to tell the function what value we want to display. We will learn more about how functions work and how to create our own in later episodes.

We can display multiple things at once using only one `print` call:

PYTHON

```
print(patient_id, 'weight in kilograms:', weight_kg)
```

OUTPUT

```
inflam_001 weight in kilograms: 60.3
```

We can also call a function inside of another function call. For example, Python has a built-in function called `type` that tells you a value's data type:

PYTHON

```
print(type(60.3))
print(type(patient_id))
```

OUTPUT

```
<class 'float'>
<class 'str'>
```

Moreover, we can do arithmetic with variables right inside the `print` function:

PYTHON

```
print('weight in pounds:', 2.2 * weight_kg)
```

OUTPUT

```
weight in pounds: 132.66
```

The above command, however, did not change the value of `weight_kg`:

PYTHON

```
print(weight_kg)
```

OUTPUT

```
60.3
```

To change the value of the `weight_kg` variable, we have to **assign** `weight_kg` a new value using the equals `=` sign:

PYTHON

```
weight_kg = 65.0
print('weight in kilograms is now:', weight_kg)
```

OUTPUT

```
weight in kilograms is now: 65.0
```

SEEING DATA TYPES

What are the data types of the following variables?

PYTHON

```
planet = 'Earth'
apples = 5
distance = 10.5
```

SEEING DATA TYPES

What are the data types of the following variables?

PYTHON

```
planet = 'Earth'
apples = 5
distance = 10.5
```

PYTHON

```
print(type(planet))
print(type(apples))
print(type(distance))
```

OUTPUT

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

CHECK YOUR UNDERSTANDING

What values do the variables `mass` and `age` have after each of the following statements? Test your answer by executing the lines.

PYTHON

```
mass = 47.5
age = 122
mass = mass * 2.0
age = age - 20
```

CHECK YOUR UNDERSTANDING

What values do the variables `mass` and `age` have after each of the following statements? Test your answer by executing the lines.

```
mass = 47.5
age = 122
mass = mass * 2.0
age = age - 20
```

PYTHON

```
`mass` holds a value of 47.5, `age` does not exist
`mass` still holds a value of 47.5, `age` holds a value of 12
`mass` now has a value of 95.0, `age`'s value is still 122
`mass` still has a value of 95.0, `age` now holds 102
```

Lists and Strings

Python lists

We create a list by putting values inside square brackets and separating the values with commas:

PYTHON

```
odds = [1, 3, 5, 7]
print('odds are:', odds)
```

OUTPUT

```
odds are: [1, 3, 5, 7]
```

We can access elements of a list using indices – numbered positions of elements in the list. These positions are numbered starting at 0, so the first element has an index of 0.

PYTHON

```
print('first element:', odds[0])
print('last element:', odds[3])
print('"-1" element:', odds[-1])
```

OUTPUT

```
first element: 1
last element: 7
"-1" element: 7
```

There is one important difference between lists and strings: we can change the values in a list, but we cannot change individual characters in a string. For example:

PYTHON

```
names = ['Curie', 'Darwing', 'Turing']  # typo in Darwin's name
print('names is originally:', names)
names[1] = 'Darwin'  # correct the name
print('final value of names:', names)
```

OUTPUT

```
names is originally: ['Curie', 'Darwing', 'Turing']
final value of names: ['Curie', 'Darwin', 'Turing']
```

PYTHON

```
name = 'Darwin'
name[0] = 'd'
```

ERROR

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
<ipython-input-8-220df48aeb2e> in <module>()
      1 name = 'Darwin'
----> 2 name[0] = 'd'

TypeError: 'str' object does not support item assignment
```

Lists are mutable,
Strings are immutable!

OVERLOADING

`+` usually means addition, but when used on strings or lists, it means “concatenate”. Given that, what do you think the multiplication operator `*` does on lists? In particular, what will be the output of the following code?

PYTHON

```
counts = [2, 4, 6, 8, 10]
repeats = counts * 2
print(repeats)
```

1. `[2, 4, 6, 8, 10, 2, 4, 6, 8, 10]`
2. `[4, 8, 12, 16, 20]`
3. `[[2, 4, 6, 8, 10],[2, 4, 6, 8, 10]]`
4. `[2, 4, 6, 8, 10, 4, 8, 12, 16, 20]`

The technical term for this is *operator overloading*: a single operator, like `+` or `*`, can do different things depending on what it’s applied to.

OVERLOADING

`+` usually means addition, but when used on strings or lists, it means “concatenate”. Given that, what do you think the multiplication operator `*` does on lists? In particular, what will be the output of the following code?

PYTHON

```
counts = [2, 4, 6, 8, 10]
repeats = counts * 2
print(repeats)
```

1. `[2, 4, 6, 8, 10, 2, 4, 6, 8, 10]`
2. `[4, 8, 12, 16, 20]`
3. `[[2, 4, 6, 8, 10],[2, 4, 6, 8, 10]]`
4. `[2, 4, 6, 8, 10, 4, 8, 12, 16, 20]`

The technical term for this is *operator overloading*: a different things depending on what it’s applied to.

The multiplication operator `*` used on a list replicates elements of the list and concatenates them together:

OUTPUT

```
[2, 4, 6, 8, 10, 2, 4, 6, 8, 10]
```

It’s equivalent to:

PYTHON

```
counts + counts
```

HETEROGENEOUS LISTS

Lists in Python can contain elements of different types. Example:

PYTHON

```
sample_ages = [10, 12.5, 'Unknown']
```

NESTED LISTS

Since a list can contain any Python variables, it can even contain other lists.

For example, you could represent the products on the shelves of a small grocery shop as a nested list called `veg`:

To store the contents of the shelf in a nested list, you write it this way:

PYTHON

```
veg = [['lettuce', 'lettuce', 'peppers', 'zucchini'],
       ['lettuce', 'lettuce', 'peppers', 'zucchini'],
       ['lettuce', 'cilantro', 'peppers', 'zucchini']]
```

To reference a specific basket on a specific shelf, you use two indexes. The first index represents the row (from top to bottom) and the second index represents the specific basket (from left to right).

PYTHON

```
print(veg[0][0])
```

OUTPUT

```
'lettuce'
```

Subsets of lists and strings can be accessed by specifying ranges of values in brackets, similar to how we accessed ranges of positions in a NumPy array. This is commonly referred to as “slicing” the list/string.

PYTHON

```
binomial_name = 'Drosophila melanogaster'
group = binomial_name[0:10]
print('group:', group)

species = binomial_name[11:23]
print('species:', species)

chromosomes = ['X', 'Y', '2', '3', '4']
autosomes = chromosomes[2:5]
print('autosomes:', autosomes)

last = chromosomes[-1]
print('last:', last)
```

OUTPUT

```
group: Drosophila
species: melanogaster
autosomes: ['2', '3', '4']
last: 4
```

Loops and repetition

An example task that we might want to repeat is accessing numbers in a list, which we will do by printing each number on a line of its own.

PYTHON

```
odds = [1, 3, 5, 7]
```

In Python, a list is basically an ordered collection of elements, and every element has a unique number associated with it — its index. This means that we can access elements in a list using their indices. For example, we can get the first number in the list `odds`, by using `odds[0]`. One way to print each number is to use four `print` statements:

PYTHON

```
print(odds[0])
print(odds[1])
print(odds[2])
print(odds[3])
```

OUTPUT

```
1
3
5
7
```

This is a bad approach for three reasons:

1. **Not scalable**. Imagine you need to print a list that has hundreds of elements. It might be easier to type them in manually.

2. **Difficult to maintain**. If we want to decorate each printed element with an asterisk or any other character, we would have to change four lines of code. While this might not be a problem for small lists, it would definitely be a problem for longer ones.

3. **Fragile**. If we use it with a list that has more elements than what we initially envisioned, it will only display part of the list's elements. A shorter list, on the other hand, will cause an error because it will be trying to display elements of the list that do not exist.

PYTHON

```
odds = [1, 3, 5]
print(odds[0])
print(odds[1])
print(odds[2])
print(odds[3])
```

OUTPUT

```
1
3
5
```

ERROR

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call las
<ipython-input-3-7974b6cdaf14> in <module>()
      3 print(odds[1])
      4 print(odds[2])
----> 5 print(odds[3])

IndexError: list index out of range
```

Back
To Top

Here's a better approach: a for loop

PYTHON

```
odds = [1, 3, 5, 7]
for num in odds:
    print(num)
```

Notice the indentation!!

OUTPUT

```
1
3
5
7
```

This is shorter — certainly shorter than something that prints every number in a hundred-number list — and more robust as well:

PYTHON

```
odds = [1, 3, 5, 7, 9, 11]
for num in odds:
    print(num)
```

OUTPUT

```
1
3
5
7
9
11
```

WHAT'S IN A NAME?

In the example above, the loop variable was given the name `num` as a mnemonic; it is short for 'number'. We can choose any name we want for variables. We might just as easily have chosen the name `banana` for the loop variable, as long as we use the same name when we invoke the variable inside the loop:

PYTHON

```
odds = [1, 3, 5, 7, 9, 11]
for banana in odds:
    print(banana)
```

OUTPUT

```
1
3
5
7
9
11
```

It is a good idea to choose variable names that are meaningful, otherwise it would be more difficult to understand what the loop is doing.

Here's another loop that repeatedly updates a variable:

PYTHON

```
length = 0
names = ['Curie', 'Darwin', 'Turing']
for value in names:
    length = length + 1
print('There are', length, 'names in the list.')
```

OUTPUT

```
There are 3 names in the list.
```

Note that a loop variable is a variable that is being used to record progress in a loop. It still exists after the loop is over, and we can re-use variables previously defined as loop variables as well:

PYTHON

```
name = 'Rosalind'
for name in ['Curie', 'Darwin', 'Turing']:
    print(name)
print('after the loop, name is', name)
```

OUTPUT

```
Curie
Darwin
Turing
after the loop, name is Turing
```

Note also that finding the length of an object is such a common operation that Python actually has a built-in function to do it called `len`:

PYTHON

```
print(len([0, 1, 2, 3]))
```

OUTPUT

```
4
```

FROM 1 TO N

Python has a built-in function called `range` that generates a sequence of numbers. `range` can accept 1, 2, or 3 parameters.

- If one parameter is given, `range` generates a sequence of that length, starting at zero and incrementing by 1. For example, `range(3)` produces the numbers `0, 1, 2`.
- If two parameters are given, `range` starts at the first and ends just before the second, incrementing by one. For example, `range(2, 5)` produces `2, 3, 4`.
- If `range` is given 3 parameters, it starts at the first one, ends just before the second one, and increments by the third one. For example, `range(3, 10, 2)` produces `3, 5, 7, 9`.

Using `range`, write a loop that uses `range` to print the first 3 natural numbers:

FROM 1 TO N

Python has a built-in function called `range` that generates a sequence of numbers. `range` can accept 1, 2, or 3 parameters.

- If one parameter is given, `range` generates a sequence of that length, starting at zero and incrementing by 1. For example, `range(3)` produces the numbers `0, 1, 2`.
- If two parameters are given, `range` starts at the first and ends just before the second, incrementing by one. For example, `range(2, 5)` produces `2, 3, 4`.
- If `range` is given 3 parameters, it starts at the first one, ends just before the second one, and increments by the third one. For example, `range(3, 10, 2)` produces `3, 5, 7, 9`.

Using `range`, write a loop tha

PYTHON

```
for number in range(1, 4):
    print(number)
```

UNDERSTANDING THE LOOPS

Given the following loop:

PYTHON

```
word = 'oxygen'
for letter in word:
    print(letter)
```

How many times is the body of the loop executed?

- 3 times
- 4 times
- 5 times
- 6 times

UNDERSTANDING THE LOOPS

Given the following loop:

PYTHON

```
word = 'oxygen'
for letter in word:
    print(letter)
```

How many times is the body of the loop executed?

- 3 times
- 4 times
- 5 times
- 6 times

The body of the loop is executed 6 times.

Choice and “Control Flow”

Conditionals

We can ask Python to take different actions, depending on a condition, with an `if` statement:

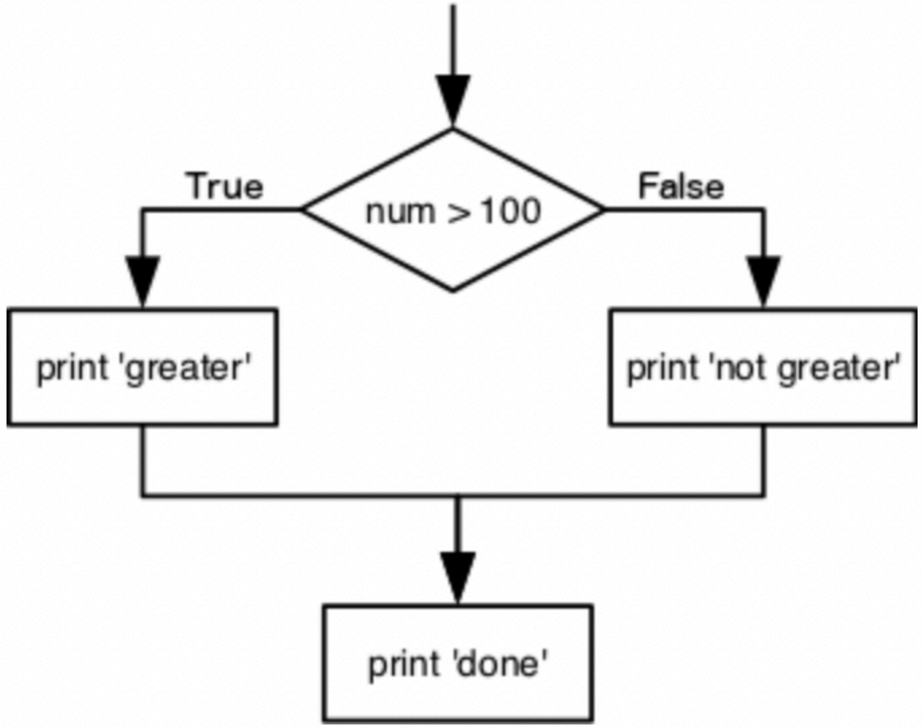
PYTHON

```
num = 37
if num > 100:
    print('greater')
else:
    print('not greater')
print('done')
```

OUTPUT

```
not greater
done
```

The second line of this code uses the keyword `if` to tell Python that we want to make a choice. If the test that follows the `if` statement is true, the body of the `if` (i.e., the set of lines indented underneath it) is executed, and “greater” is printed. If the test is false, the body of the `else` is executed instead, and “not greater” is printed. Only one or the other is ever executed before continuing on with program execution to print “done”:



Conditional statements don't have to include an `else`. If there isn't one, Python simply does nothing if the test is false:

PYTHON

```
num = 53
print('before conditional...')
if num > 100:
    print(num, 'is greater than 100')
print('...after conditional')
```

OUTPUT

```
before conditional...
...after conditional
```

We can also chain several tests together using `elif`, which is short for “else if”. The following Python code uses `elif` to print the sign of a number.

PYTHON

```
num = -3

if num > 0:
    print(num, 'is positive')
elif num == 0:
    print(num, 'is zero')
else:
    print(num, 'is negative')
```

OUTPUT

```
-3 is negative
```

Note that to test for equality we use a double equals sign `==` rather than a single equals sign `=` which is used to assign values.

COMPARING IN PYTHON

Along with the `>` and `==` operators we have already used for comparing values in our conditionals, there are a few more options to know about:

- `>`: greater than
- `<`: less than
- `==`: equal to
- `!=`: does not equal
- `>=`: greater than or equal to
- `<=`: less than or equal to

We can also combine tests using `and` and `or`. `and` is only true if both parts are true:

PYTHON

```
if (1 > 0) and (-1 >= 0):
    print('both parts are true')
else:
    print('at least one part is false')
```

OUTPUT

```
at least one part is false
```

while `or` is true if at least one part is true:

PYTHON

```
if (1 < 0) or (1 >= 0):
    print('at least one test is true')
```

OUTPUT

```
at least one test is true
```

`True` AND `False`

`True` and `False` are special words in Python called `booleans`, which represent truth values. A statement such as `1 < 0` returns the value `False`, while `-1 < 0` returns the value `True`.

HOW MANY PATHS?

Consider this code:

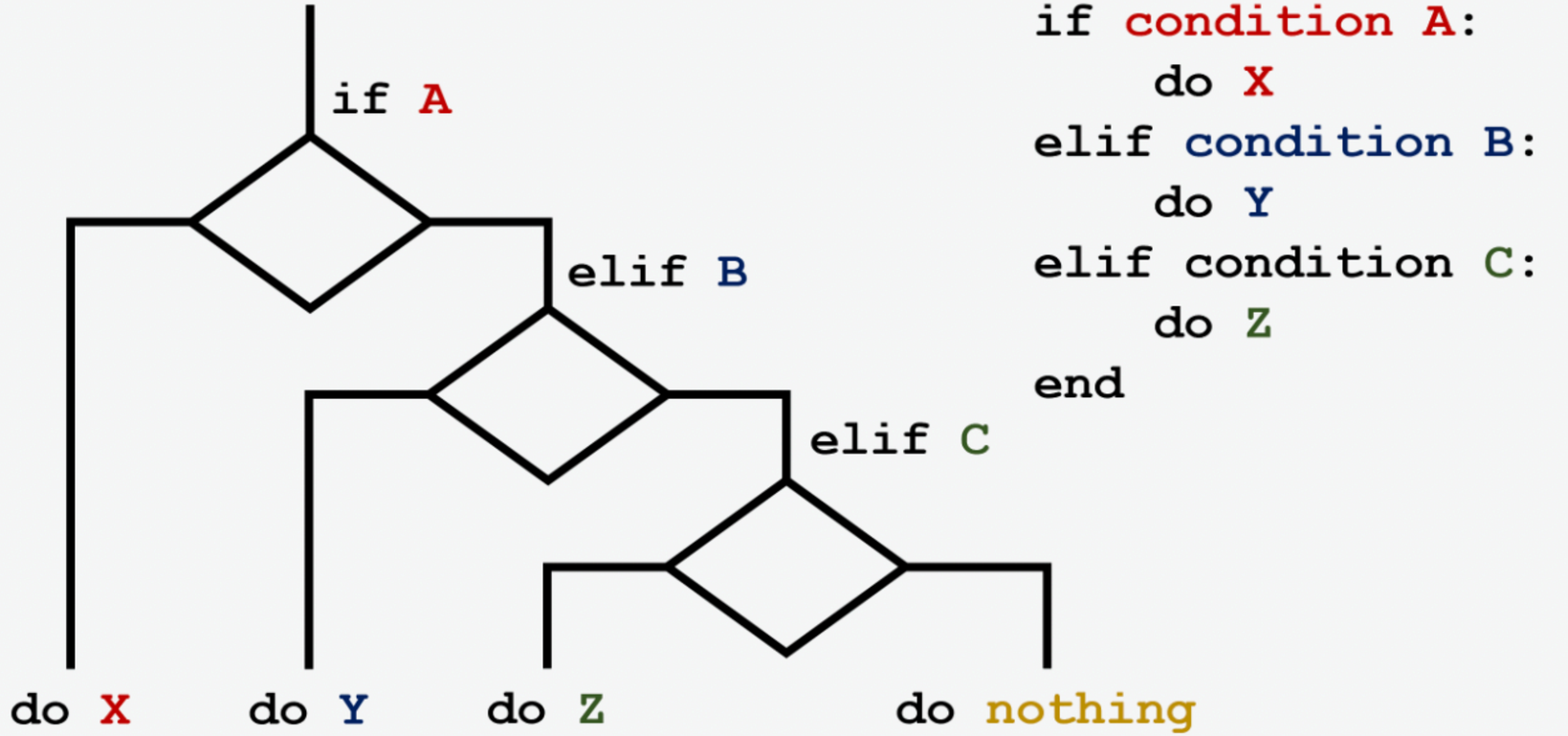
PYTHON

```python
if 4 > 5:
    print('A')
elif 4 == 5:
    print('B')
elif 4 < 5:
    print('C')
```

Which of the following would be printed if you were to run this code? Why did you pick this answer?

1. A
2. B
3. C
4. B and C

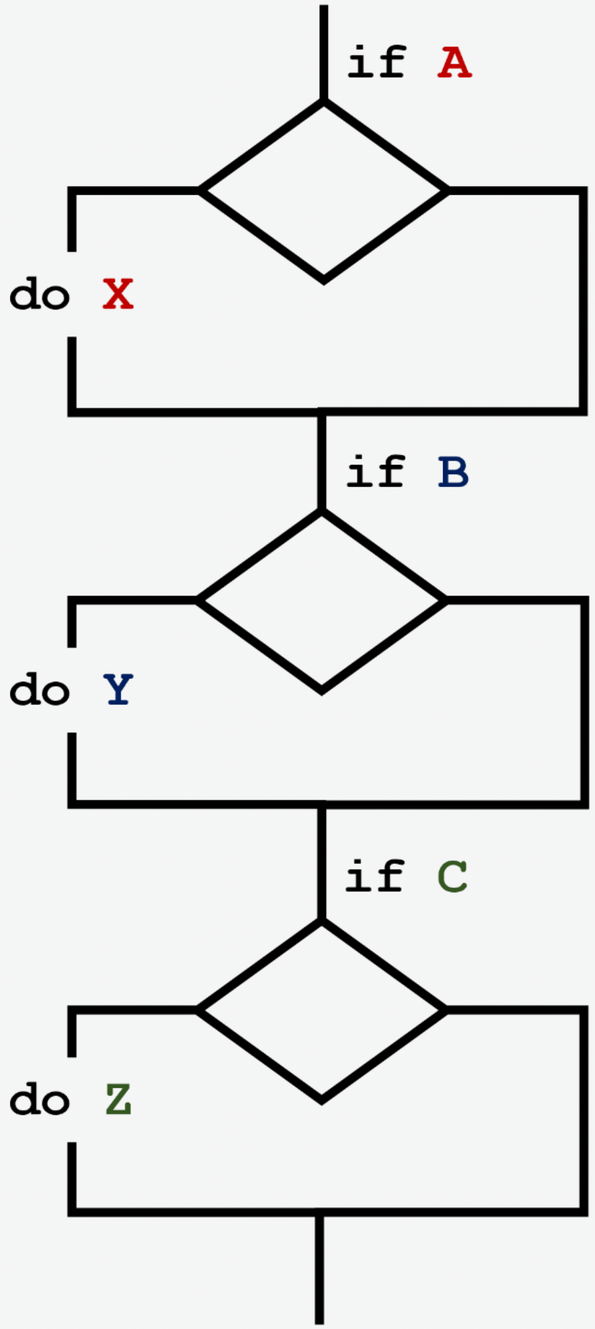
C gets printed because the first two conditions, `4 > 5` and `4 == 5`, are not true, but `4 < 5` is true. In this case only one of these conditions can be true for at a time, but in other scenarios multiple `elif` conditions could be met. In these scenarios only the action associated with the first true `elif` condition will occur, starting from the top of the conditional section.




```
if condition A:
    do X
elif condition B:
    do Y
elif condition C:
    do Z
end
```

A true B true → do X C false	A true B false → do X C true	A false B true → do Y C true

This contrasts with the case of multiple `if` statements, where every action can occur as long as their condition is met.




```
if condition A:
    do X
if condition B:
    do Y
if condition C:
    do Z
```

A true B true C false	→	do X do Y
A true B false C true	→	do X do Z
A false B true C true	→	do Y do Z

Functions and Code Reuse

At this point, we've seen that code can have Python make decisions about what it sees in our data. What if we want to convert some of our data, like taking a temperature in Fahrenheit and converting it to Celsius. We could write something like this for converting a single number

PYTHON

```
fahrenheit_val = 99
celsius_val = ((fahrenheit_val - 32) * (5/9))
```

and for a second number we could just copy the line and rename the variables

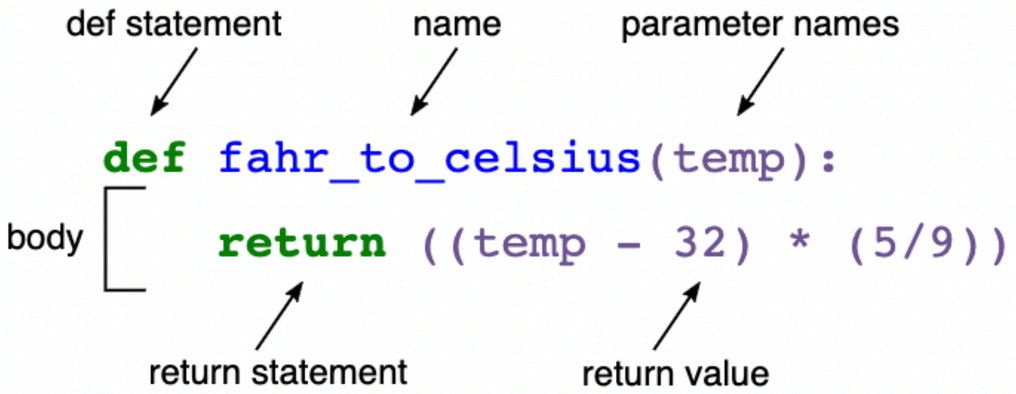
PYTHON

```
fahrenheit_val = 99
celsius_val = ((fahrenheit_val - 32) * (5/9))

fahrenheit_val2 = 43
celsius_val2 = ((fahrenheit_val2 - 32) * (5/9))
```

But we would be in trouble as soon as we had to do this more than a couple times. Cutting and pasting it is going to make our code get very long and very repetitive, very quickly. We'd like a way to package our code so that it is easier to reuse, a shorthand way of re-executing longer pieces of code. In Python we can use 'functions'. Let's start by defining a function `fahr_to_celsius` that converts temperatures from Fahrenheit to Celsius:

PYTHON

```
def explicit_fahr_to_celsius(temp):
    # Assign the converted value to a variable
    converted = ((temp - 32) * (5/9))
    # Return the value of the new variable
    return converted

def fahr_to_celsius(temp):
    # Return converted value more efficiently using the return
    # function without creating a new variable. This code does
    # the same thing as the previous function but it is more explicit
    # in explaining how the return command works.
    return ((temp - 32) * (5/9))
```



The function definition opens with the keyword `def` followed by the name of the function (`fahr_to_celsius`) and a parenthesized list of parameter names (`temp`). The body of the function — the statements that are executed when it runs — is indented below the definition line. The body concludes with a `return` keyword followed by the return value.

When we call the function, the values we pass to it are assigned to those variables so that we can use them inside the function. Inside the function, we use a return statement to send a result back to whoever asked for it.

Let's try running our function.

PYTHON

```
fahr_to_celsius(32)
```

This command should call our function, using "32" as the input and return the function value.

In fact, calling our own function is no different from calling any other function:

PYTHON

```
print('freezing point of water:', fahr_to_celsius(32), 'C')
print('boiling point of water:', fahr_to_celsius(212), 'C')
```

OUTPUT

```
freezing point of water: 0.0 C
boiling point of water: 100.0 C
```

Composing Functions

Now that we've seen how to turn Fahrenheit into Celsius, we can also write the function to turn Celsius into Kelvin:

PYTHON

```
def celsius_to_kelvin(temp_c):
    return temp_c + 273.15

print('freezing point of water in Kelvin:', celsius_to_kelvin(0.))
```

OUTPUT

```
freezing point of water in Kelvin: 273.15
```

What about converting Fahrenheit to Kelvin? We could write out the formula, but we don't need to. Instead, we can compose the two functions we have already created:

PYTHON

```
def fahr_to_kelvin(temp_f):
    temp_c = fahr_to_celsius(temp_f)
    temp_k = celsius_to_kelvin(temp_c)
    return temp_k

print('boiling point of water in Kelvin:', fahr_to_kelvin(212.0))
```

OUTPUT

```
boiling point of water in Kelvin: 373.15
```

What about converting Fahrenheit to Kelvin? We could write out the formula, but we don't need to. Instead, we can compose the two functions we have already created:

PYTHON

```
def fahr_to_kelvin(temp_f):
    temp_c = fahr_to_celsius(temp_f)
    temp_k = celsius_to_kelvin(temp_c)
    return temp_k

print('boiling point of water in Kelvin:', fahr_to_kelvin(212.0))
```

OUTPUT

```
boiling point of water in Kelvin: 373.15
```

PYTHON

```
print('Again, temperature in Kelvin was:', temp_k)
```

Variable Scope

In composing our temperature conversion functions, we created variables inside of those functions, `temp`, `temp_c`, `temp_f`, and `temp_k`. We refer to these variables as local variables because they no longer exist once the function is done executing. If we try to access their values outside of the function, we will encounter an error:

PYTHON

```
print('Again, temperature in Kelvin was:', temp_k)
```

ERROR

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call las
<ipython-input-1-eed2471d229b> in <module>
----> 1 print('Again, temperature in Kelvin was:', temp_k)

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

If you want to reuse the temperature in Kelvin after you have calculated it with `fahr_to_kelvin`, you can store the result of the function call in a variable:

PYTHON

```
temp_kelvin = fahr_to_kelvin(212.0)
print('temperature in Kelvin was:', temp_kelvin)
```

OUTPUT

```
temperature in Kelvin was: 373.15
```

The variable `temp_kelvin`, being defined outside any function, is said to be global.

Inside a function, one can read the value of such global variables:

PYTHON

```
def print_temperatures():
  print('temperature in Fahrenheit was:', temp_fahr)
  print('temperature in Kelvin was:', temp_kelvin)

temp_fahr = 212.0
temp_kelvin = fahr_to_kelvin(temp_fahr)

print_temperatures()
```

OUTPUT

```
temperature in Fahrenheit was: 212.0
temperature in Kelvin was: 373.15
```

Readable functions

Consider these two functions:

PYTHON

```
def s(p):
    a = 0
    for v in p:
        a += v
    m = a / len(p)
    d = 0
    for v in p:
        d += (v - m) * (v - m)
    return numpy.sqrt(d / (len(p) - 1))

def std_dev(sample):
    sample_sum = 0
    for value in sample:
        sample_sum += value

    sample_mean = sample_sum / len(sample)

    sum_squared_devs = 0
    for value in sample:
        sum_squared_devs += (value - sample_mean) * (value - sample_mean)

    return numpy.sqrt(sum_squared_devs / (len(sample) - 1))
```

MIXING DEFAULT AND NON-DEFAULT PARAMETERS

Given the following code:

PYTHON

```
def numbers(one, two=2, three, four=4):
    n = str(one) + str(two) + str(three) + str(four)
    return n

print(numbers(1, three=3))
```

what do you expect will be printed? What is actually printed? What rule do you think Python is following?

1. `1234`
2. `one2three4`
3. `1239`
4. `SyntaxError`

MIXING DEFAULT AND NON-DEFAULT PARAMETERS

Given the following code:

PYTHON

```python
def numbers(one, two=2, three, four=4):
    n = str(one) + str(two) + str(three) + str(four)
    return n

print(numbers(1, three=3))
```

what do you expect will be printed? What is actually printed? What rule do you think Python is following?

1. `1234`
2. `one2three4`
3. `1239`
4. `SyntaxError`

PYTHON < >

```
def func(a, b=3, c=6):
    print('a: ', a, 'b: ', b, 'c:', c)

func(-1, 2)
```

1. `a: b: 3 c: 6`
2. `a: -1 b: 3 c: 6`
3. `a: -1 b: 2 c: 6`
4. `a: b: -1 c: 2`

PYTHON

```
def func(a, b=3, c=6):
    print('a: ', a, 'b: ', b, 'c:', c)

func(-1, 2)
```

1. `a: b: 3 c: 6`
2. `a: -1 b: 3 c: 6`
3. `a: -1 b: 2 c: 6`
4. `a: b: -1 c: 2`

The given call to `func` displays `a: -1 b: 2 c: 6`. -1 is assigned to the first parameter `a`, 2 is assigned to the next parameter `b`, and `c` is not passed a value, so it uses its default value 6.

Why Functions

- **Encapsulation:**
  - A function is often useful if it encapsulates or represents a single idea. For example, the function which converts Fahrenheit to Celsius is one idea that we contained in a single function.
  - By encapsulating the idea, we **reuse code**. For example, what if there was a typo in the function? We go back and modify the one line of code which defines the function. If we copied and pasted, we go back and modify many, many lines of code and we may miss some, leading to inconsistent use.

- **Abstraction:**
  - Any user can call the function to convert between Fahrenheit and Celsius without knowing the implementation. That is, without knowing exactly what the lines of code say

Errors and Exceptions

Errors in Python have a very specific form, called a traceback. Let's examine one:

PYTHON

```
# This code has an intentional error. You can type it directly or
# use it for reference to understand the error message below.
def favorite_ice_cream():
    ice_creams = [
        'chocolate',
        'vanilla',
        'strawberry'
    ]
    print(ice_creams[3])

favorite_ice_cream()
```

ERROR

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call las
<ipython-input-1-70bd89baa4df> in <module>()
      9     print(ice_creams[3])
      10
----> 11 favorite_ice_cream()

<ipython-input-1-70bd89baa4df> in favorite_ice_cream()
      7         'strawberry'
      8     ]
----> 9     print(ice_creams[3])
      10
      11 favorite_ice_cream()

IndexError: list index out of range
```

Outer level

Inner level

This particular traceback has two levels. You can determine the number of levels by looking for the number of arrows on the left hand side. In this case:

1. The first shows code from the cell above, with an arrow pointing to Line 11 (which is `favorite_ice_cream()`).
2. The second shows some code in the function `favorite_ice_cream`, with an arrow pointing to Line 9 (which is `print(ice_creams[3])`).

The last level is the actual place where the error occurred. The other level(s) show what function the program executed to get to the next level down. So, in this case, the program first performed a function call to the function `favorite_ice_cream`. Inside this function, the program encountered an error on Line 6, when it tried to run the code `print(ice_creams[3])`.

PYTHON

```python
# This code has an intentional error. Do not type it directly
# use it for reference to understand the error message below.
def print_message(day):
    messages = [
        'Hello, world!',
        'Today is Tuesday!',
        'It is the middle of the week.',
        'Today is Donnerstag in German!',
        'Last day of the week!',
        'Hooray for the weekend!',
        'Aw, the weekend is almost over.'
    ]
    print(messages[day])

def print_sunday_message():
    print_message(7)

print_sunday_message()
```

What do you think will happen?

PYTHON

```python
# This code has an intentional error. Do not type it direct
# use it for reference to understand the error message belo
def print_message(day):
    messages = [
        'Hello, world!',
        'Today is Tuesday!',
        'It is the middle of the week.',
        'Today is Donnerstag in German!',
        'Last day of the week!',
        'Hooray for the weekend!',
        'Aw, the weekend is almost over.'
    ]
    print(messages[day])

def print_sunday_message():
    print_message(7)

print_sunday_message()
```

ERROR

```
---------------------------------------------------------------
IndexError                                Traceback (most rec
<ipython-input-7-3ad455d81842> in <module>
     16     print_message(7)
     17
---> 18 print_sunday_message()
     19

<ipython-input-7-3ad455d81842> in print_sunday_message()
     14
     15 def print_sunday_message():
---> 16     print_message(7)
     17
     18 print_sunday_message()

<ipython-input-7-3ad455d81842> in print_message(day)
     11         'Aw, the weekend is almost over.'
     12     ]
---> 13     print(messages[day])
     14
     15 def print_sunday_message():

IndexError: list index out of range
```

PYTHON

```python
# This code has an intentional error. Do not type it direct
# use it for reference to understand the error message belo
def print_message(day):
    messages = [
        'Hello, world!',
        'Today is Tuesday!',
        'It is the middle of the week.',
        'Today is Donnerstag in German!',
        'Last day of the week!',
        'Hooray for the weekend!',
        'Aw, the weekend is almost over.'
    ]
    print(messages[day])

def print_sunday_message():
    print_message(7)

print_sunday_message()
```

ERROR

```
---------------------------------------------------------------
IndexError                                Traceback (most rec
<ipython-input-7-3ad455d81842> in <module>
     16     print_message(7)
     17
---> 18 print_sunday_message()
     19

<ipython-input-7-3ad455d81842> in print_sunday_message()
     14
     15 def print_sunday_message():
---> 16     print_message(7)
     17
     18 print_sunday_message()

<ipython-input-7-3ad455d81842> in print_message(day)
     11         'Aw, the weekend is almost over.'
     12     ]
---> 13     print(messages[day])
     14
     15 def print_sunday_message():

IndexError: list index out of range
```

1. How many levels does the traceback have?
2. What is the function name where the error occurred?
3. On which line number in this function did the error occur?
4. What is the type of error?
5. What is the error message?

PYTHON

```
1  # This code has an intentional error. Do not type it direct
2  # use it for reference to understand the error message belo
3  def print_message(day):
4      messages = [
5          'Hello, world!',
6          'Today is Tuesday!',
7          'It is the middle of the week.',
8          'Today is Donnerstag in German!',
9          'Last day of the week!',
10         'Hooray for the weekend!',
11         'Aw, the weekend is almost over.'
12     ]
13     print(messages[day])
14
15 def print_sunday_message():
16     print_message(7)
17
18 print_sunday_message()
```

ERROR

```
---------------------------------------------------------------
IndexError                                Traceback (most rec
<ipython-input-7-3ad455d81842> in <module>
     16     print_message(7)
     17
1 ---> 18 print_sunday_message()
     19

<ipython-input-7-3ad455d81842> in print_sunday_message()
     14
     15 def print_sunday_message():
2 ---> 16     print_message(7)
     17
     18 print_sunday_message()

<ipython-input-7-3ad455d81842> in print_message(day)
     11         'Aw, the weekend is almost over.'
     12     ]
3 ---> 13     print(messages[day])
     14
     15 def print_sunday_message():

IndexError: list index out of range
```

1. How many levels does the traceback have?
2. What is the function name where the error occurred?
3. On which line number in this function did the error occur?
4. What is the type of error?
5. What is the error message?

PYTHON

```
1  # This code has an intentional error. Do not type it direct
2  # use it for reference to understand the error message belo
3  def print_message(day):
4      messages = [
5          'Hello, world!',
6          'Today is Tuesday!',
7          'It is the middle of the week.',
8          'Today is Donnerstag in German!',
9          'Last day of the week!',
10         'Hooray for the weekend!',
11         'Aw, the weekend is almost over.'
12     ]
13     print(messages[day])
14
15 def print_sunday_message():
16     print_message(7)
17
18 print_sunday_message()
```

ERROR

```
---------------------------------------------------------------
IndexError                                Traceback (most rec
<ipython-input-7-3ad455d81842> in <module>
     16     print_message(7)
     17
---> 18 print_sunday_message()
     19

<ipython-input-7-3ad455d81842> in print_sunday_message()
     14
     15 def print_sunday_message():
---> 16     print_message(7)
     17
     18 print_sunday_message()

<ipython-input-7-3ad455d81842> in print_message(day)
     11         'Aw, the weekend is almost over.'
     12     ]
---> 13     print(messages[day])
     14
     15 def print_sunday_message():

IndexError: list index out of range
```

1. How many levels does the traceback have?
2. What is the function name where the error occurred?
3. On which line number in this function did the error occur?
4. What is the type of error?
5. What is the error message?

PYTHON

```python
# This code has an intentional error. Do not type it direct
# use it for reference to understand the error message belo
def print_message(day):
    messages = [
        'Hello, world!',
        'Today is Tuesday!',
        'It is the middle of the week.',
        'Today is Donnerstag in German!',
        'Last day of the week!',
        'Hooray for the weekend!',
        'Aw, the weekend is almost over.'
    ]
    print(messages[day])

def print_sunday_message():
    print_message(7)

print_sunday_message()
```

ERROR

```
---------------------------------------------------------------
IndexError                                Traceback (most rec
<ipython-input-7-3ad455d81842> in <module>
     16     print_message(7)
     17
---> 18 print_sunday_message()
     19

<ipython-input-7-3ad455d81842> in print_sunday_message()
     14
     15 def print_sunday_message():
---> 16     print_message(7)
     17
     18 print_sunday_message()

<ipython-input-7-3ad455d81842> in print_message(day)
     11         'Aw, the weekend is almost over.'
     12     ]
---> 13     print(messages[day])
     14
     15 def print_sunday_message():

IndexError: list index out of range
```

1. How many levels does the traceback have?
2. What is the function name where the error occurred?
3. On which line number in this function did the error occur?
4. What is the type of error?
5. What is the error message?

PYTHON

```python
# This code has an intentional error. Do not type it direct
# use it for reference to understand the error message belo
def print_message(day):
    messages = [
        'Hello, world!',
        'Today is Tuesday!',
        'It is the middle of the week.',
        'Today is Donnerstag in German!',
        'Last day of the week!',
        'Hooray for the weekend!',
        'Aw, the weekend is almost over.'
    ]
    print(messages[day])

def print_sunday_message():
    print_message(7)

print_sunday_message()
```

ERROR

```
---------------------------------------------------------------
IndexError                                Traceback (most rec
<ipython-input-7-3ad455d81842> in <module>
     16     print_message(7)
     17
---> 18 print_sunday_message()
     19

<ipython-input-7-3ad455d81842> in print_sunday_message()
     14
     15 def print_sunday_message():
---> 16     print_message(7)
     17
     18 print_sunday_message()

<ipython-input-7-3ad455d81842> in print_message(day)
     11         'Aw, the weekend is almost over.'
     12     ]
---> 13     print(messages[day])
     14
     15 def print_sunday_message():

IndexError: list index out of range
```

1. How many levels does the traceback have?
2. What is the function name where the error occurred?
3. On which line number in this function did the error occur?
4. What is the type of error?
5. What is the error message?

PYTHON

```python
# This code has an intentional error. Do not type it direct
# use it for reference to understand the error message belo
def print_message(day):
    messages = [
        'Hello, world!',
        'Today is Tuesday!',
        'It is the middle of the week.',
        'Today is Donnerstag in German!',
        'Last day of the week!',
        'Hooray for the weekend!',
        'Aw, the weekend is almost over.'
    ]
    print(messages[day])

def print_sunday_message():
    print_message(7)

print_sunday_message()
```

ERROR

```
---------------------------------------------------------------
IndexError                                Traceback (most rec
<ipython-input-7-3ad455d81842> in <module>
     16     print_message(7)
     17
---> 18 print_sunday_message()
     19

<ipython-input-7-3ad455d81842> in print_sunday_message()
     14
     15 def print_sunday_message():
---> 16     print_message(7)
     17
     18 print_sunday_message()

<ipython-input-7-3ad455d81842> in print_message(day)
     11         'Aw, the weekend is almost over.'
     12     ]
---> 13     print(messages[day])
     14
     15 def print_sunday_message():

IndexError: list index out of range
```

1. How many levels does the traceback have?
2. What is the function name where the error occurred?
3. On which line number in this function did the error occur?
4. What is the type of error?
5. What is the error message?

Syntax Errors

- **An algorithm for making a sandwich might begin:**
  - 1. Put the peanut butter on the bread

Syntax Errors

- **An algorithm for making a sandwich might begin:**
  - 1. Put the peanut butter on the bread

Syntax Errors

- **An algorithm for making a sandwich might begin:**
  - 1. **Take the peanut butter from the jar with a knife and** put the peanut butter on the bread

Syntax Errors

- **An algorithm for making a sandwich might begin:**
  - 1. **Take the peanut butter from the jar with a knife and** put the peanut butter on the bread
  - 2. Put the two pieces of bread together

- **It may feel like the computer is ignoring you, but actually it is trying its best to follow your instructions *exactly.* Remember, computers are DUMB!**

Syntax Errors

PYTHON < >

```
def some_function()
    msg = 'hello, world!'
    print(msg)
     return msg
```

Syntax Errors

PYTHON < >

```
def some_function()
    msg = 'hello, world!'
    print(msg)
     return msg
```

ERROR < >

```
  File "<ipython-input-3-6bb841ea1423>", line 1
    def some_function()
                      ^
SyntaxError: invalid syntax
```

Syntax Errors

PYTHON < >

```
def some_function():
    msg = 'hello, world!'
    print(msg)
     return msg
```

Syntax Errors

PYTHON

```
def some_function():
    msg = 'hello, world!'
    print(msg)
     return msg
```

ERROR

```
  File "<ipython-input-4-ae290e7659cb>", line 4
    return msg
    ^
IndentationError: unexpected indent
```

Variable Name Error

PYTHON < >

```
print(a)
```

ERROR < >

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call las
<ipython-input-7-9d7b17ad5387> in <module>()
----> 1 print(a)

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

What is wrong with this code?

PYTHON < >

```
for number in range(10):
    # use a if the number is a multiple of 3, otherwise use b
    if (Number % 3) == 0:
        message = message + a
    else:
        message = message + 'b'
print(message)
```

What is wrong with this code?

PYTHON < >

```
for number in range(10):
    # use a if the number is a multiple of 3, otherwise use b
    if (Number % 3) == 0:
        message = message + a
    else:
        message = message + 'b'
print(message)
```

Hint: 3 things

What is wrong with this code?

PYTHON

```
for number in range(10):
    # use a if the number is a multiple of 3, otherwise use b
    if (Number % 3) == 0:
        message = message + a
    else:
        message = message + 'b'
print(message)
```

PYTHON

```
message = ''
for number in range(10):
    # use a if the number is a multiple of 3, otherwise use b
    if (number % 3) == 0:
        message = message + 'a'
    else:
        message = message + 'b'
print(message)
```

Libraries

Libraries

- Most of the time, when you want to write code to do a task, someone else has already written some similar code and uploaded it to the web
- You can install and import these libraries and use them in your code
- There are 2 main libraries used in the code for this course: numpy and matplotlib

Numpy

- You can write a function to get linearly spaced numbers in a list:

```
>>> def linear_list(n=1):
...     return [i for i in range(n)]
...
>>> linear_list(4)
[0, 1, 2, 3]
```

- Or, you can call a numpy function:

```
>>> import numpy as np
>>> np.linspace(0,3,4)
array([0., 1., 2., 3.])
```

- Note that these two implementations give different results! The numpy function returns an “array”, which we have not seen before!

Numpy arrays

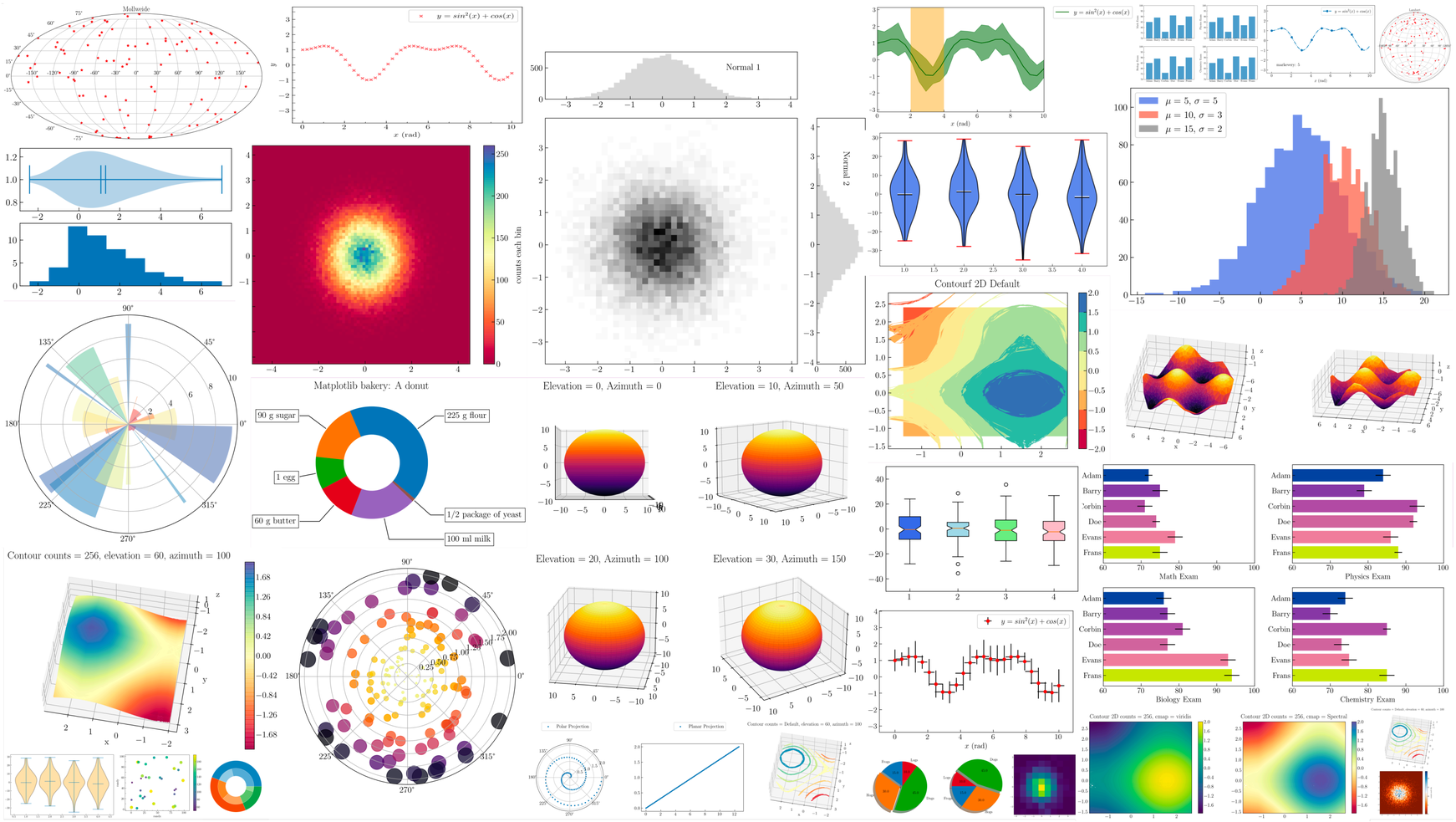
- Similar to lists, but with some notably different properties

```
>>> [1.0, 2.0, 3.0] + [1.0, 2.0, 3.0]
[1.0, 2.0, 3.0, 1.0, 2.0, 3.0]
>>>
>>>
>>> np.linspace(0,2,3) + np.linspace(0,2,3)
array([0., 2., 4.])
```

- Numpy arrays add the values when we use “+”, lists concatenate and create a longer list when we use “+”
- In general, numpy arrays have more useful implementations for mathematics and linear algebra than lists do.
- In this course, we will generally tell you exactly which numpy functions you need to use when appropriate

Matplotlib

- Implements many functions which allow you to make all sorts of different plots
- Again, we will tell you which functions you need when you need to make a plot



Jupyter Notebooks

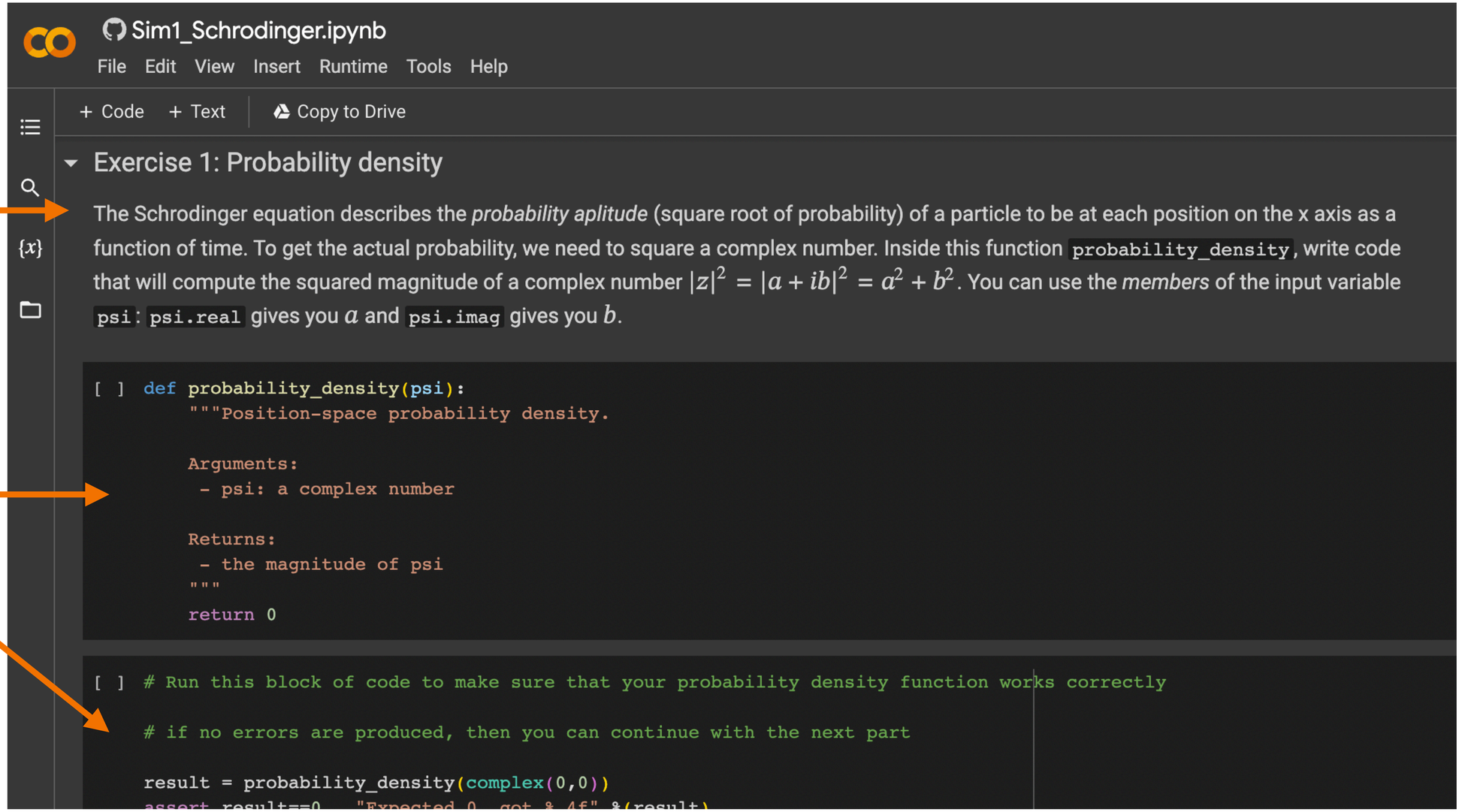
Jupyter Notebooks

- **This course will use notebooks, which allow code and text/images to be placed together and hosted online.**

Text block

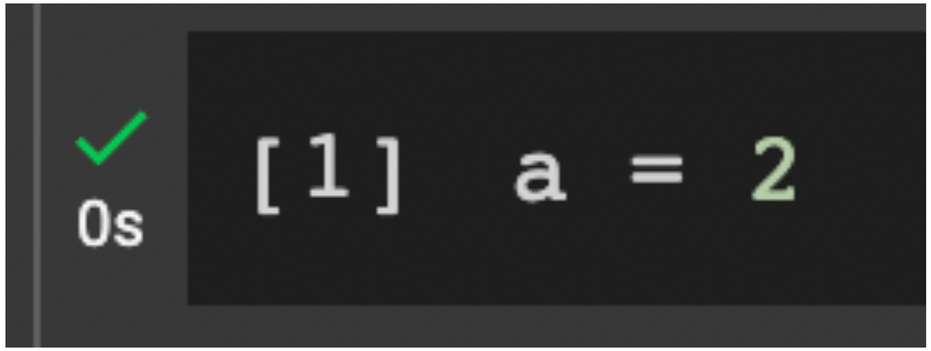
Code blocks

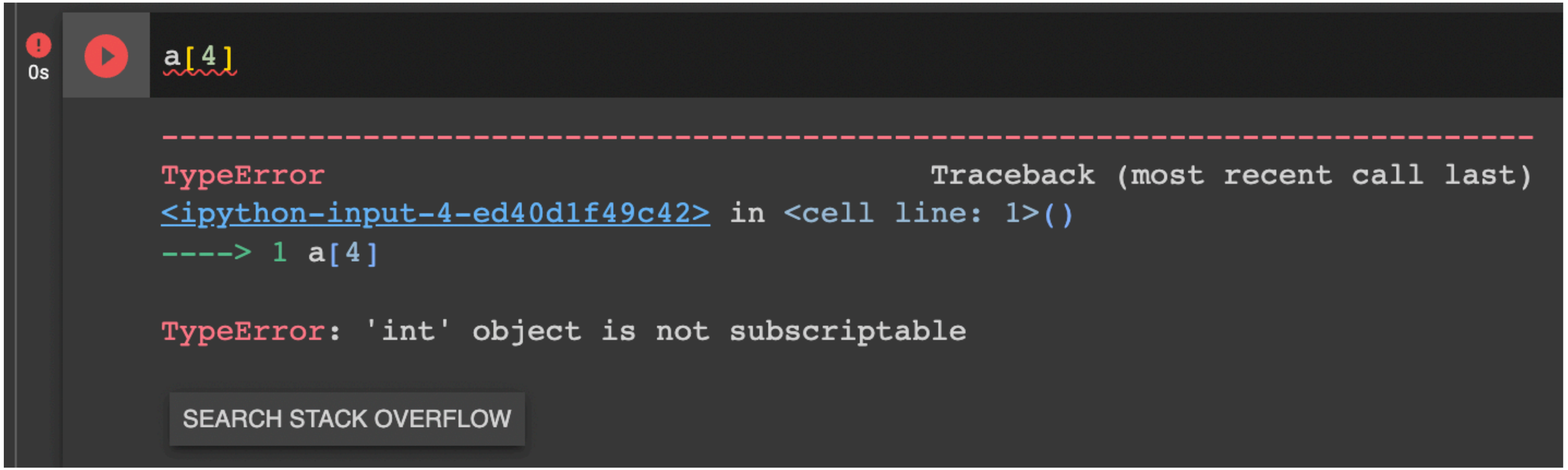
Run code by
Selecting block
And hit enter



Code Blocks

- When you run a code block, a green check mark appears
- If an error occurs, a red exclamation point will appear





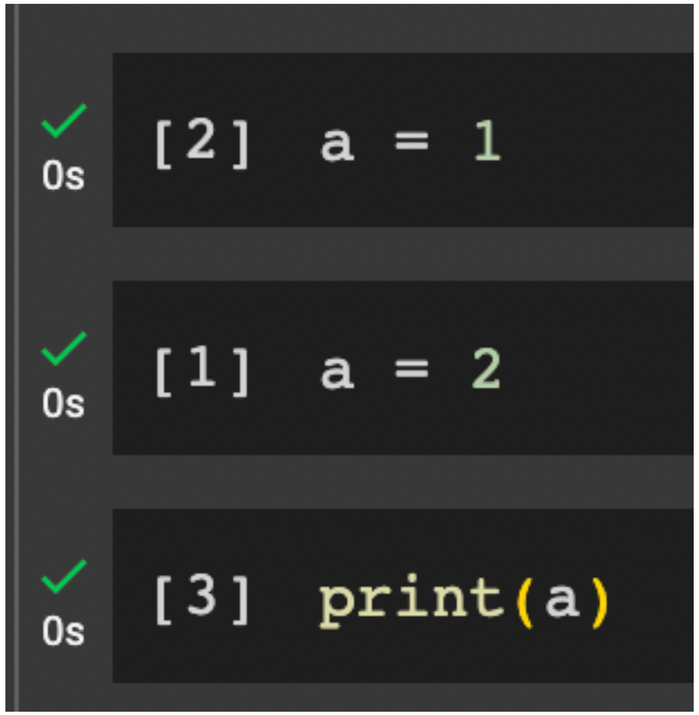
Code Blocks

- What is the output from each of these two “algorithms”

Python Interpreter

```
>>> a = 1
>>> a = 2
>>> print(a)
```

Jupyter Notebook



Code Blocks

- What is the output from each of these two “algorithms”

Python Interpreter

```
>>> a = 1
>>> a = 2
>>> print(a)
2
```

Jupyter Notebook

```
[2]  a = 1
```

0s

```
[1]  a = 2
```

0s

```
[3]  print(a)
```

0s

```
1
```

Huh???

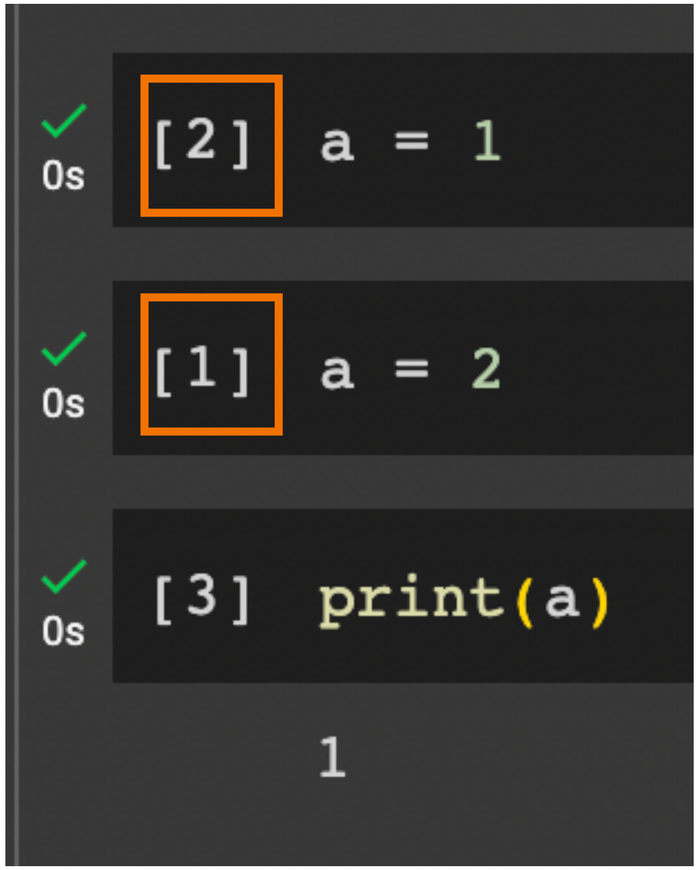
Code Blocks

- What is the output from each of these two “algorithms”

Python Interpreter

```
>>> a = 1
>>> a = 2
>>> print(a)
2
```

Jupyter Notebook

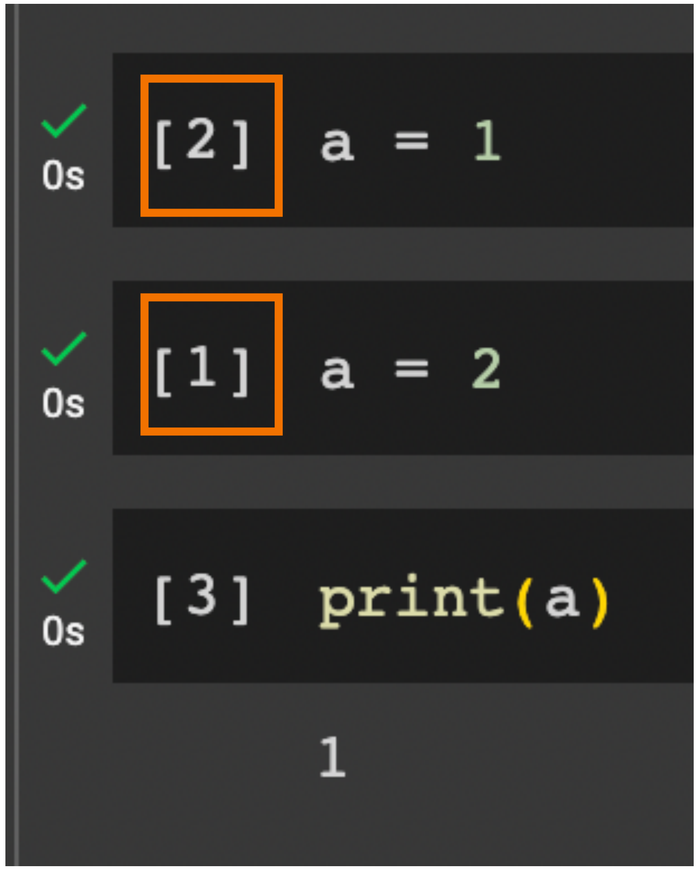


Jupyter Notebooks

- The most confusing part of notebooks: our code blocks can be executed out of order!!!

- This may cause an unexpected result and, most confusingly, no errors!!!

Jupyter Notebook



Jupyter Notebooks

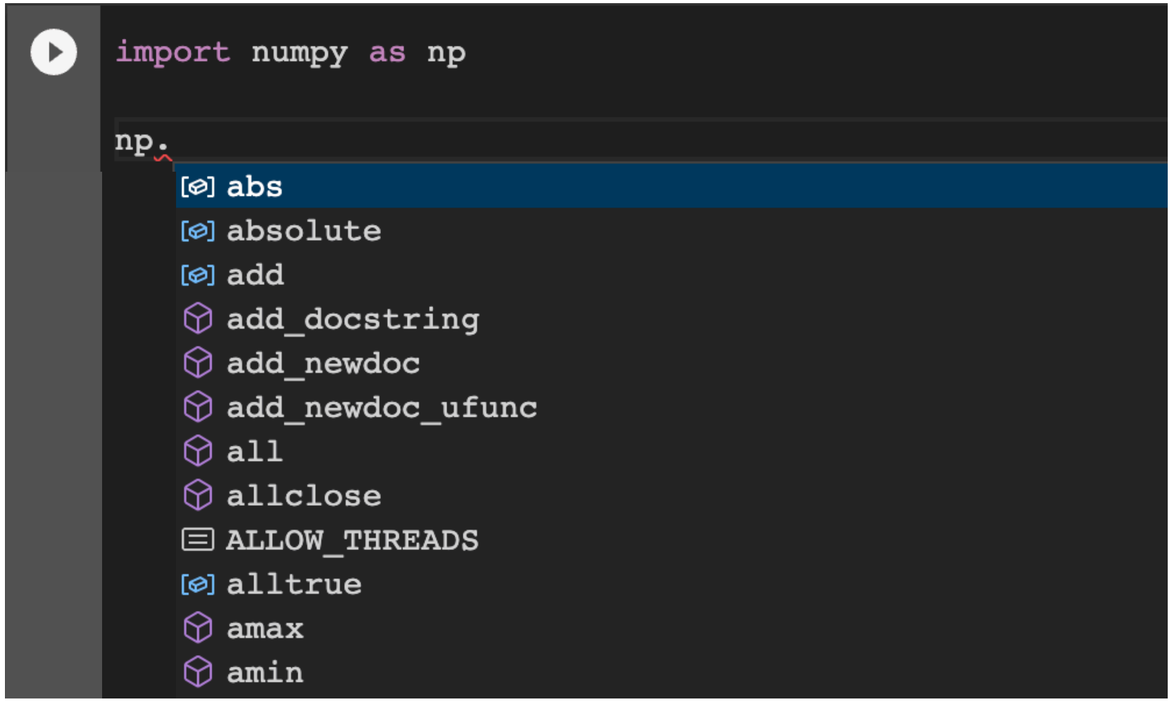
- The most confusing part of notebooks: our code blocks can be executed out of order!!!
- This may cause an unexpected result and, most confusingly, no errors!!!

Jupyter Notebook

```
0s   [2]  a = 1

0s   [1]  a = 2

0s   [3]  print(a)

     1
```

Notebooks and Libraries

- Notebooks aren't *all* bad, they do have many important libraries automatically available
- They also give nice autocomplete when using library functions.
- This is very helpful if you can't remember exactly the name of a function.



Conclusions

- **In these lectures, we learned about:**
  - Variables, values, and expressions
  - Lists and strings
  - Loops and repetition
  - Choice and “Control Flow”
  - Functions and code reuse
  - Errors
  - Libraries
  - Jupyter Notebooks

- **This gave you the absolute basics of how to write algorithms in python, use existing libraries, and share your code using Jupyter notebooks.**

- **To truly learn python, you will need to spend *years* learning programming. Don’t worry if you are still struggling throughout these two weeks! Ask for help!**