Introduction to Python

Karolos POTAMIANOS

University of Oxford (UK)

March 2, 2022

European School in Instrumentation for Particle and Astroparticle Physics (ESIPAP)

European Scientific Institute, Archamps, France

(ONLINE)

A bit about me …

karolos.potamianos@cern.ch

@kpotamianos

http://linkedin.com/in/karolos

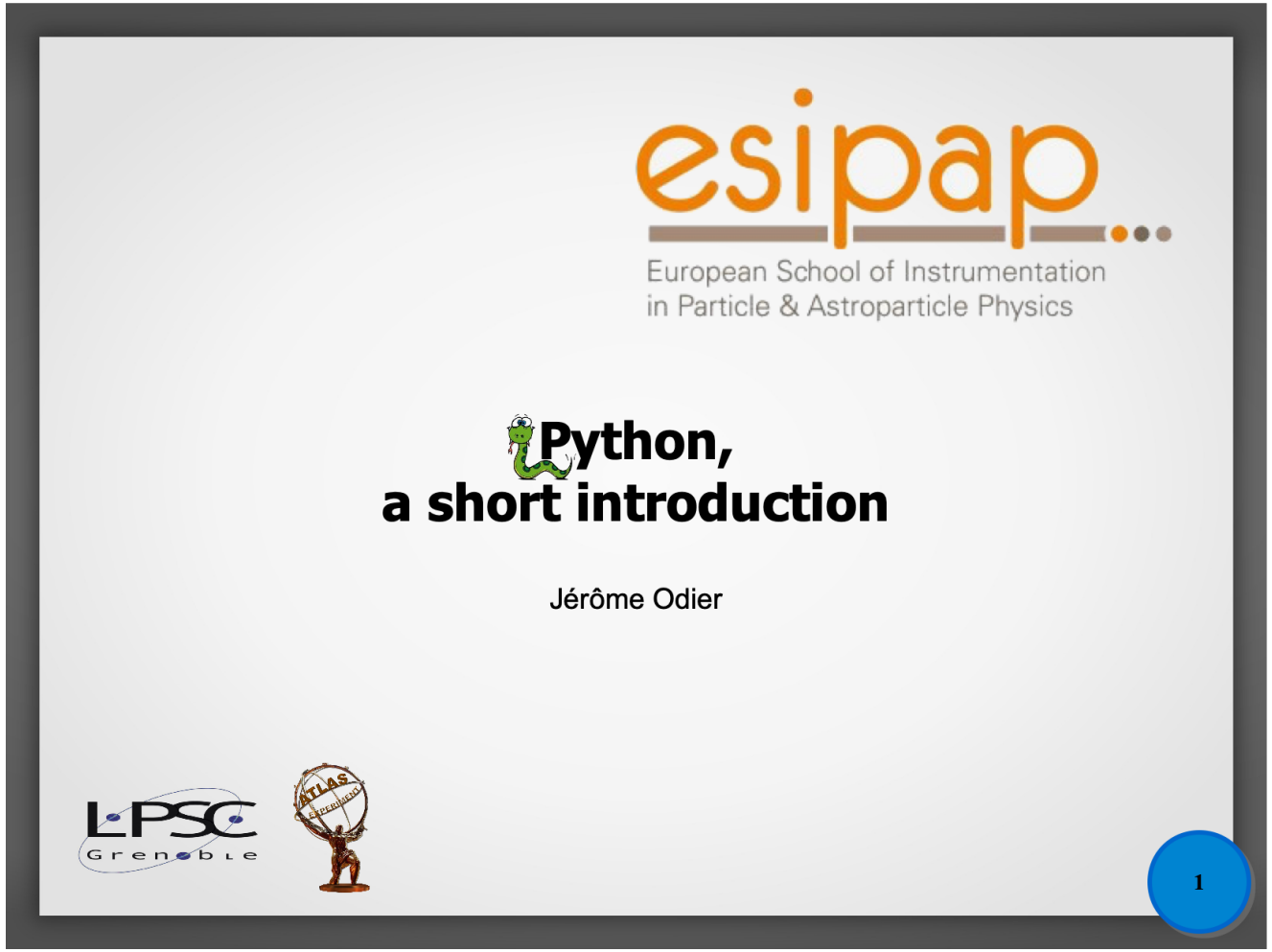
Education

- BSc./MSc. in Applied Physics
  @ Ecole Polytechnique, ULB, Belgium
- BSc. in Business Administration
  @ Solvay Business School, ULB, Belgium
- PhD in Particle Physics
  @ Purdue University, USA

Research

- 2012 – Present :: Researcher
  @ CERN
- 2012 – 2017 :: Postdoctoral Research Fellow
  @ Lawrence Berkeley National Laboratory (LBNL), USA
- 2017 – 2020 :: Research Fellow
  @ Deutsches Elektronen-Synchrotron (DESY), GER
- 2020 – Present :: Ernest Rutherford Fellow
  @ University of Oxford, UK

Some of the material is inspired from past ESIPAP lectures by Jérôme Odier, whom I thank for allowing me to re-use.



Why Python ?

- Why Python ? Why (yet) another programming language ?

"Python is an **easy to learn**, **powerful programming language**. It has **efficient high-level data structures** and a **simple but effective approach to object-oriented programming**. Python's elegant syntax and **dynamic typing**, together with its **interpreted nature**, make it an ideal language for **scripting and rapid application development** in many areas **on most platforms**."

"The Python interpreter is easily **extended with new functions and data types implemented in C or C++ (or other languages callable from C).**"

Python

Appeared in	1991; 30 years ago
Designed by	Guido van Rossum
Stable release	3.10.2 ~~(& 2.7.18)~~
URL	http://www.python.org/
OS	cross-platform

Why Python ?

- Python is very nice for thigs like
  - writing scripts / command line tools (e.g. replacing bash)
  - symbolic computation
  - data analysis
- Its interpreted nature means that it (the language) is not meant for high-performance tasks (though it gets better at some of it)
  - unless it calls dedicated specialized functions in C/C++ (or other languages)
  - luckily this integration is possible and quite easy
- Python is a great tool for scripting and benefits from a huge ecosystem of libraries and tools

```
SymPy
pyROOT
NumPy
SciPy
Matplotlib
...
```

The Philosophy of Python – The Zen of Python

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.

And 10 more rules…

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

https://www.python.org/dev/peps/pep-0020/

More about Python

- **Python is**
  - structured (if, for, etc.)
  - object-oriented
  - module-oriented

  - modern (reflexion and garbage collection)
  - cross-plaform (portable code)
  - interpreted (bytecode virtual machine, like Java)
  - **not optimized** for performance (but can wrap around such code)

What is Python ?

- Python is a **backend programming language** that's great for beginners.
- Python is **approachable**. Even if you haven't taken a CS class, you can still write a useful tool in Python. It's **high-level**, so you don't have to deal with the lower-level aspects of programming, such as memory management.
- Python can be used for scripting, web scraping, and creating data sets. It's popular in the scientific community for **scientific computing**; there are libraries that make it easy to share academic code projects in Python.
- Python is a **web programming language**, so it interfaces with the internet. It knows how to receive and send web requests and talk to databases.
- Python is said to be "**loosely typed**." This category of programming languages doesn't require you to state the type of value a function returns when you define the function or the type of variable before you create it.
- The Python **community is welcoming**, well-maintained, and well-documented. That's important for a beginner!

[Source]

Indentation in Python

- One of Van Rossum's decisions was to make **indentation meaningful**
  - **This is unusual in programming languages.**
  - Despite critics, this feature is part of the reason it is **both readable and popular**.
  - Good code style and readability is enforced by the way you must write Python.

C++

```
void myFunction() {
    /* function body */
}
```

Python

```
def myFunction():
    # The function body
```

Why Python is Good for Beginners

- Python **syntax is very similar to English**, so it’s intuitive, which helps you understand it.
  - You don't have to look up what symbols mean when you use Python.

Using the interpreter

```
import random


def get_random_color():
    colors = ['green', 'blue', 'red', 'yello']
    random_color = random.choice(colors)
    return random_color
```

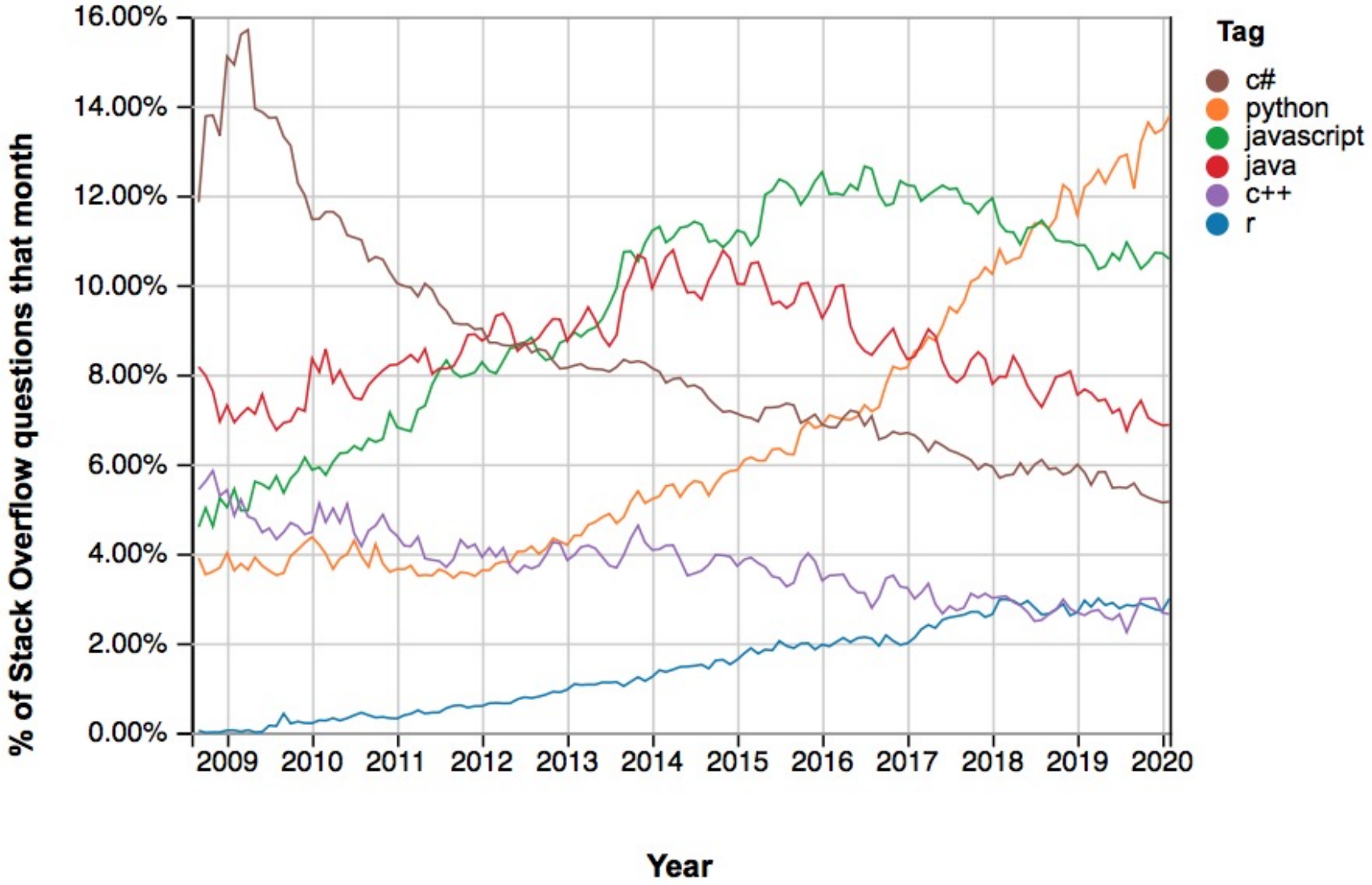
Who can guess what this function is doing ?

Disadvantages of Python

- Python is **slower** than other languages.
  - **Trade off** between how **high-level** and **abstract** a programming language is and how **efficient** it is in terms of **speed, memory usage** and **space usage**.
  - **It is not low-level**, and not as fast or efficient as a compiled, lower-level language.
  - It's **less common to use Python** to build distributed database systems or other systems **where speed is incredibly important**.
- There are also some concerns about **scalability**, although you can make Python scalable with different implementations of the language, such as PyPy.

- BUT it's probably nothing you should worry about unless you develop applications for high-performance computing or time-critical applications (e.g. data acquisition)
- REMEMBER that **readability counts**, and that it most of the time doesn't matter whether your code takes 1 second rather than 10 ms (assuming it doesn't have to run repeatedly, e.g. for many events) and **human time is more precious than CPU time.**

Usage of Python



Source: [Stack Overflow Trends]

The versions of Python

- Perhaps the most confusing par about python is that version 3 is not backward compatible with version 2
  - Python 3 started as a cleanup which ended up changing too many things
  - Decision to use Unicode by default was the lead cause (as the rest could have been done using the **deprecation** process)
- Nevertheless, there is a high usage of v2 together with a large community (driven by machine learning) using the new features of v3
  - Many packages are maintained for both v2 and v3
- **BUT Python 2 reached End-Of-Life (EOL) in 2020**
  - I'd recommend you **focus on Python 3** (but remember v2 will stick around)
- More info (in case you're curious):
  - Why Python 3: https://snarky.ca/why-python-3-exists/
  - Porting from v2 to v3: https://docs.python.org/3.7/howto/pyporting.html

Let's get into Python

The Python Console

Using the interpreter

```
$ python3
Python 3.7.7 (default, Mar 10 2020, 15:43:03)
[Clang 11.0.0 (clang-1100.0.33.17)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> print("Hello world!")
Hello world!
>>> quit()
$
```

Calling a script

```
$ cat !$
cat hello.py
#!/usr/bin/env python3

# Going to print out something
print("Hello world!")
$ python3 hello.py
Hello world!
$ chmod +x hello.py && ./hello.py
Hello world!
```

Python Data Structures

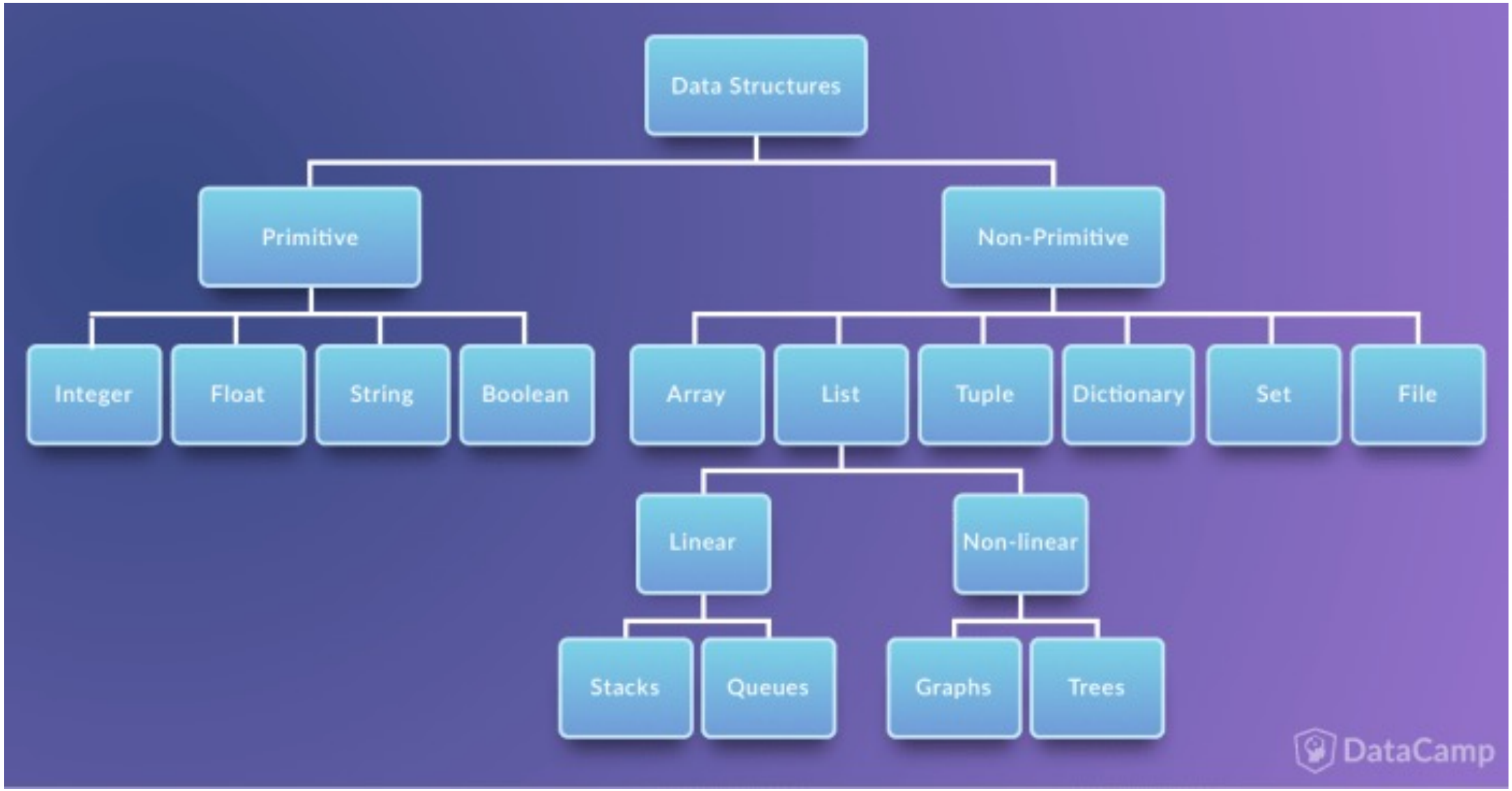


Image: DataCamp

(Python) Primitive Data Structures

- **Integers:** represent **numeric data**, and more specifically, whole numbers from negative infinity to infinity, like 4, 5, or -1.
- **Float:** stands for **'floating point number'**. You can use it for rational numbers, usually ending with a decimal figure, such as 1.11 or 3.14.
- **String:** collections of alphabets, words or other characters. In Python, you can create strings by enclosing a sequence of characters within a **pair of single or double quotes**. For example: 'cake', "cookie", etc.
- **Boolean:** built-in data type that can take up the values **True** or **False**, which often makes them interchangeable with the integers 1 and 0. Booleans are **useful in conditional and comparison expressions**.

Operators

- Like every programming language, Python has **operators** to perform operations on **data types**
- Like in mathematics, there is a priority in the **order** in which the operations are **executed**
- **How much is 4 * 3 + 1 ? 13 or 16 ?**
- **Parentheses can be used to explicitly ensure which order was meant, e.g., (4*3) + 1 vs. 4 * (3+1)**
- **Many bugs** due to misremembering the **priority of operators**

Operator	Description
()	Parentheses (grouping)
f(args...)	Function call
x[index:index]	Slicing
x[index]	Subscription
x.attribute	Attribute reference
**	Exponentiation
~*x*	Bitwise not
+*x*, -*x*	Positive, negative
*, /, %	Multiplication, division, remainder
+, -	Addition, subtraction
<<, >>	Bitwise shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
in, not in, is, is not, <, <=, >, >=,	Comparisons, membership, identity
<>, !=, ==	
not *x*	Boolean NOT
and	Boolean AND
or	Boolean OR
lambda	Lambda expression

Operations on Primitive Data Structures

Operations on Float

```
# Floats
x = 4.0
y = 2.0

print(x + y)   # Addition
print(x - y)   # Subtraction
print(x * y)   # Multiplication
print(x / y)   # Returns the quotient
print(x % y)   # Returns the remainder
print(abs(x)) # Absolute value
print(x ** y) # x to the power y
```

In Python, you do **not have to explicitly state the type of the variable or your data**. That is because it is a **dynamically typed language**. Such languages are the those where the type of data an object can store is **mutable**.

Operations on Primitive Data Structures

Operations on Float

```
>>> # Floats
>>> x = 4.0
>>> y = 2.0
>>>
>>> print(x + y)   # Addition
6.0
>>> print(x - y)   # Subtraction
2.0
>>> print(x * y)   # Multiplication
8.0
>>> print(x / y)   # Returns the quotient
2.0
>>> print(x % y)   # Returns the remainder
0.0
>>> print(abs(x)) # Absolute value
4.0
>>> print(x ** y) # x to the power y
16.0
```

Operations on Primitive Data Structures

Operations on Strings

```
>>> x = 'Cake'
>>> y = 'Cookie'
>>> x + ' & ' + y
'Cake & Cookie'
>>> x * 2
'CakeCake'
>>> x[2:] # A string is basically an array of characters
'ke'
>>> y[0] + y[1]
'Co'
>>> a = '4' # Character 4, not the digit 4
>>> b = '2' # Character 2, not the digit 2
>>> a + b
'42'
```

Operations on Primitive Data Structures

Operations on Strings

```
>>> str.capitalize('cookie')
'Cookie'
>>> str1 = "Cake 4 U"
>>> str2 = "404"
>>> len(str1)
8
>>> str1.isdigit()
False
>>> str2.isdigit()
True
>>> str1.replace('4 U', str2)
'Cake 404'
>>> str1 = 'cookie'
>>> str2 = 'cook'
>>> # Position where 'cook' is found in 'cookie'
>>> str1.find(str2)
0
```

Multi-line Definition of Strings

Strings

```
>>> s = """
... Hello
... This is on multiple lines"
...
... ""
... """
>>> print(s)

Hello
This is on multiple lines"

""

>>> # This is a comment and is ignored
>>>
```

Operations with Primitive Data Structures

Usage of Booleans

```
>>> x = 4
>>> y = 2
>>> x == y
False
>>> x > y
True
>>> x < y
False
```

(Implicit) Type Conversions

Implicit Type Conversions

```
>>> x = 4.0 # A float
>>> y = 2   # An integer
>>> z = x/y # Divide `x` by `y`
>>> type(z) # Check the type of `z`
<class 'float'>
>>> x = 1
>>> y = 2
>>> x/y
0.5
>>> type(x), type(y), type(x/y)
(<class 'int'>, <class 'int'>, <class 'float'>)
```

- In some cases (typically with numeric types), implicit conversions are performed (e.g. y is converted into float when doing the division)

(Explicit) Type Conversions

Explicit Type Conversions

```
>>> x = 2
>>> y = "The Godfather: Part "
>>> favorite_movie = y + x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> favorite_movie = y + str(x)
>>> print(favorite_movie)
The Godfather: Part 2
```

- In other (most) cases you'll need to perform an **explicit type conversion** (here say that we wanted to add the string representation of 2 to a string (concatenation operation)

(Python) Non-Primitive Data Structures

- **Arrays:** compact way of collecting basic data types, **all the entries in an array must be of the same data type**. They are a **more efficient way of storing a certain kind of list** (see below).
- **List:** used to store **collection of heterogeneous items**. These are **mutable** (you can change their content without changing their identity). Lists are recognizable by their square brackets **[** and **]** that hold elements, separated by a comma **,**. Lists are built into Python: no need to invoke them separately.

Python Arrays and Lists

Arrays & Lists

```
>>> import array as arr
>>> a = arr.array("I",[3,6,9])
>>> type(a)
<class 'array.array'>
>>> x = [] # Empty list
>>> type(x)
<class 'list'>
>>> x1 = [1,2,3]
>>> type(x1)
<class 'list'>
>>> x2 = list([1,'apple',3])
>>> type(x2)
<class 'list'>
>>> print(x2[1])
apple
>>> x2[1] = 'orange'
>>> print(x2)
[1, 'orange', 3]
```

More on Python Arrays

Arrays

```
>>> list_num = [1,2,45,6,7,2,90,23,435]
>>> list_char = ['c','o','o','k','i','e']
>>> list_num.append(11) # Add 11 at the end of the list
>>> print(list_num)
[1, 2, 45, 6, 7, 2, 90, 23, 435, 11]
>>> list_num.insert(0, 11)
>>> print(list_num)
[11, 1, 2, 45, 6, 7, 2, 90, 23, 435, 11]
>>> list_char.remove('o')
>>> print(list_char)
['c', 'o', 'k', 'i', 'e']
>>> list_char.pop(-2) # Removes the item at the specified position
'i'
>>> print(list_char)
['c', 'o', 'k', 'e']
>>> list_num.sort() # In-place sorting
>>> print(list_num)
[1, 2, 2, 6, 7, 11, 11, 23, 45, 90, 435]
>>> list.reverse(list_num)
>>> print(list_num)
[435, 90, 45, 23, 11, 11, 7, 6, 2, 2, 1]
```

Python Arrays vs. Lists

Arrays vs. Lists

```
>>> import array
>>> array_char = array.array("u",["c","a","t","s"])
>>> x = array_char.tostring() # not possible with list
>>> print(array_char)
array('u', 'cats')
>>> x1 = [1,2,3]
>>> x1.tostring()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'tostring'
```

Note on Arrays vs. Lists

- We can apply the `tostring()` function on the `array_char` array object because Python is aware that all the items in an array are of the same data type and hence the operation behaves the same way on each element.
- Arrays can be very useful when dealing with a large collection of homogeneous data types.
- As Python does not have to remember the data type details of each element individually; for some uses **arrays may be faster and uses less memory when compared to lists**.

Numpy Arrays

Numpy Arrays

```
>>> import numpy as np
>>> arr_a = np.array([3, 6, 9])
>>> arr_b = arr_a/3 # Performing vectorized (element-wise)
operations
>>> print(arr_b)
[1. 2. 3.]
>>> arr_ones = np.ones(4)
>>> print(arr_ones)
[1. 1. 1. 1.]
>>> multi_arr_ones = np.ones((3,4)) # Creating 2D array with
3 rows and 4 columns
>>> print(multi_arr_ones)
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

Numpy is very often use for machine learning applications (but not only!)

More Non-Primitive Data Structures

- The list data structure can be further categorised into two: **linear** and **non-linear** data structures.
  - **Stacks** and **Queues** are called *linear data structures*
  - **Graphs** and **Trees** are *non-linear data structures*
- These structures and their concepts can be relatively complex but are used extensively due to their resemblance to real world models
- In linear data structure, the data items are **organised sequentially**, or linearly. The data items are **traversed serially one after another.** All the data items in a linear structure can be traversed in a single run.
- In non-linear data structures, the data items are **not organized sequentially**. The elements could be connected to more than one element to reflect a special relationship among these items. All the items in a non-linear structure *may* not be traversed in a single run.

Stacks and Queues (and deques)

- **Stacks:** a container of objects that are inserted and removed according to the Last-In-First-Out (LIFO) concept. (Think of documents in a processing pile on a desk.)
- **Queues:** a container of objects that are inserted and removed according to the First-In-First-Out (FIFO) principle. (Think of a ticket counter where people are processed according to their arrival time.)
- **Deques:** a *double-ended queue* is a container that has the feature of adding and removing elements from either end. (Think of a history of commands that have been processed, of which you only want to remember the N most recent ones).
- These structures are used for performing many operations, from evaluating expressions to syntax parsing and algorithm scheduling.

Stacks in Python (are simply Lists)

Stacks

```
>>> # Bottom -> 1 -> 2 -> 3 -> 4 -> 5 (Top)
>>> stack = [1,2,3,4,5]
>>> stack.append(6) # Bottom -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
(Top)
>>> print(stack)
[1, 2, 3, 4, 5, 6]
>>> stack.pop() # Bottom -> 1 -> 2 -> 3 -> 4 -> 5 (Top)
6
>>> stack.pop() # Bottom -> 1 -> 2 -> 3 -> 4 (Top)
5
>>> print(stack)
[1, 2, 3, 4]
```

There are also other implementations, but this is a convenient one.

Queues in Python

Queues

```
>>> import queue
>>> q = queue.Queue()
>>> q.put(0)
>>> q.put(1)
>>> q.put(2)
>>> print(q.get(), q.get(), q.get())
0 1 2
>>>
>>> s = queue.LifoQueue() # The same class can do stacks
>>> s.put(0)
>>> s.put(1)
>>> s.put(2)
>>> print(s.get(), s.get(), s.get())
2 1 0
```

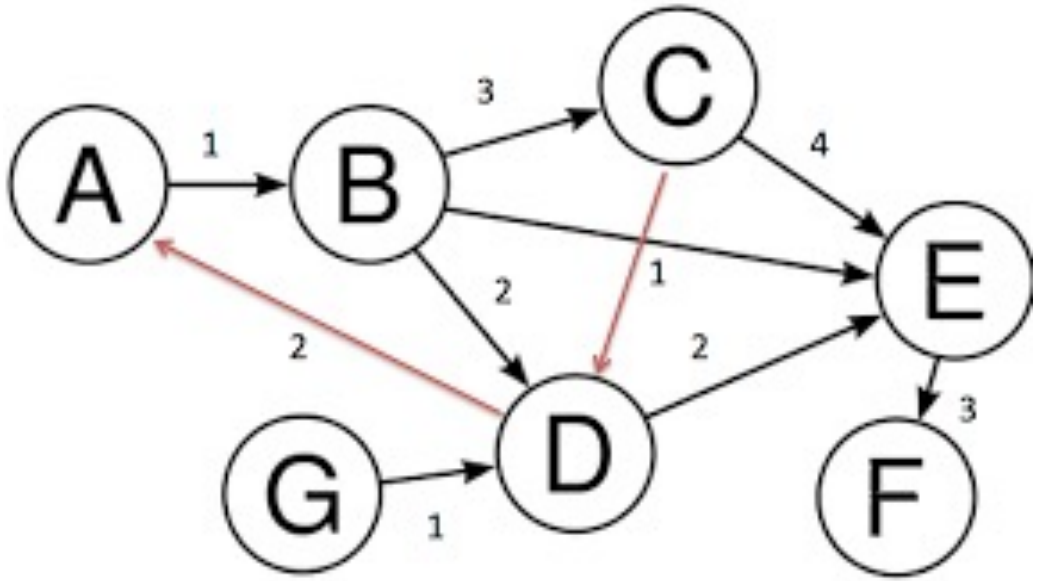
Deques in Python

Deques

```
>>> import collections
>>> d = collections.deque(["Mon","Tue","Wed"])
>>> print (d)
deque(['Mon', 'Tue', 'Wed'])
>>> d.append("Thu") # Append to the right
>>> print (d)
deque(['Mon', 'Tue', 'Wed', 'Thu'])
>>> d.appendleft("Sun") # Append to the left
>>> print (d)
deque(['Sun', 'Mon', 'Tue', 'Wed', 'Thu'])
>>> d.pop() # Remove from the right
'Thu'
>>> print (d)
deque(['Sun', 'Mon', 'Tue', 'Wed'])
>>> d.popleft() # Remove from the left
'Sun'
>>> print (d)
deque(['Mon', 'Tue', 'Wed'])
>>> d.reverse() # Reverse the dequeue
>>> print (d)
deque(['Wed', 'Tue', 'Mon'])
```

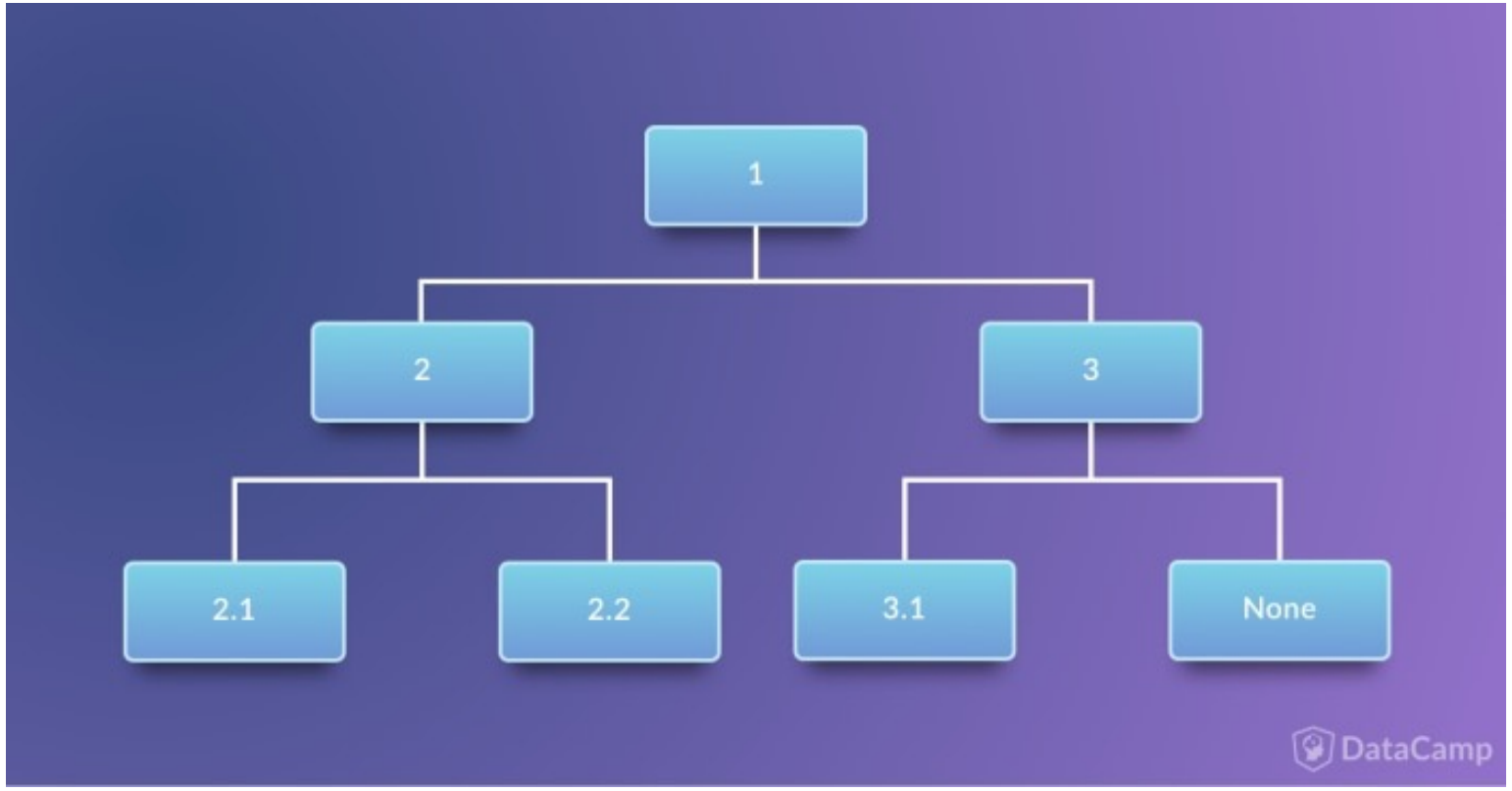
Graphs

- **Graphs:** networks consisting of **nodes**, also called vertices which may or may not be connected to each other. The lines or the path that connects two nodes is called an **edge**.
  - If the edge has a direction of flow, then it is a **directed graph**
  - If no directions are specified, it is called an **undirected graph**
  - If the edges carry a weight, it is called a **weighted graph**



Trees

- **Trees:** used to describe how data is sometimes organized, but unlike real trees, the root is on the top and the branches, leaves follow, spreading towards the bottom



Tuples

- **Tuples:** a standard sequence data type. Contrary to lists, tuples are immutable, which means once defined you cannot delete, add or edit any values inside it. This is useful in situations where you might to pass the control to code written by others, but you do not want them to manipulate data in your collection.

Tuples

```
>>> x_tuple = 1,2,3,4,5
>>> y_tuple = ('c','a','k','e')
>>> x_tuple[0]
1
>>> y_tuple[3]
'e'
>>> x_tuple[0] = 0 # Cannot change values inside a tuple
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Dictionaries

- **Dictionary:** a standard structure to represent … a dictionary (e.g. a telephone book), i.e., in cases you need to perform a lookup. They are made up **of key-value pairs**. The **key** is used to identify the item and the value holds as the name suggests, the **value** of the item.

Dictionaries

```
>>> x_dict = {'Edward':1, 'Jorge':2, 'Prem':3, 'Joe':4}
>>> del x_dict['Joe']
>>> x_dict
{'Edward': 1, 'Jorge': 2, 'Prem': 3}
>>> x_dict['Edward'] # Prints the value stored with the key
'Edward'.
1
>>> len(x_dict)
3
>>> x_dict.keys()
dict_keys(['Edward', 'Jorge', 'Prem'])
>>> x_dict.values()
dict_values([1, 2, 3])
```

Graph Implementation with Dictionary

Graph

```
>>> graph = { "a" : ["c", "d"],
...           "b" : ["d", "e"],
...           "c" : ["a", "e"],
...           "d" : ["a", "b"],
...           "e" : ["b", "c"]
...         }
>>>
>>> def define_edges(graph):
...     edges = []
...     for vertices in graph:
...         for neighbour in graph[vertices]:
...             edges.append((vertices, neighbour))
...     return edges
...
>>> print(define_edges(graph))
[('a', 'c'), ('a', 'd'), ('b', 'd'), ('b', 'e'), ('c', 'a'), ('c',
'e'), ('d', 'a'), ('d', 'b'), ('e', 'b'), ('e', 'c')]
```

Sets

- **Set:** a collection of distinct (unique) objects. These are useful to create lists that only *hold unique values in the dataset*.

Sets

```
>>> x_set = set('CAKE&COKE')
>>> y_set = set('COOKIE')
>>> print(x_set)
{'C', 'K', '&', 'A', 'E', 'O'}
>>> print(y_set) # Single unique 'o'
{'C', 'K', 'E', 'O', 'I'}
>>> print(x_set-y_set) # All the elements in x_set but
not in y_set
{'A', '&'}
>>> print(x_set|y_set) # Unique elements in x_set or
y_set or both
{'C', 'K', '&', 'A', 'E', 'O', 'I'}
```

Files

No programming language would truly be useful without the capability to store and retrieve previously stored information. Files are a common place where we hold data (there are other forms, too).

Here are some common file operations:

- `open()` to open files in your system
  - first argument is the file name, second the mode: **r**(ead), **w**(rite), **a**(ppend)
- `read()` to read entire files
- `readline()` to read one line at a time
- `write(something)` to write a `something` to a file (returns the number of characters written)
- `close()` to close the file.

None

None is frequently used in Python to represent the absence of a value, for example when default arguments are not passed to functions (more later).

The operators is and not can be used to check whether an element exists.

Sets

```
>>> x = None
>>> x is None
True
>>> x is not None
False
>>> not None
True
```

On Whitespaces and Blocks

- **Indentation** is meaningful in Python: the **same** number of spaces or tabs is needed to indent one level in the same file.
- You can use backslashes **** to go to the next line (in case of long lines)
- There are **no braces to mark blocks of code**
- **Indented blocks** have a semicolon : to start them
- Blocks must contain at least one instruction ; use **pass** if you need to make an empty block

Python Code

```
>>> print(x)
[1, 2, 3, 4, 5, 6]
>>> def printHello():
...     print("Hello")
...
>>> printHello()
Hello
```

Functions

Functions are used to modularise code and re-use the same code pieces by **calling** them again. They can have any number of arguments, provided as a comma-separated tuple.

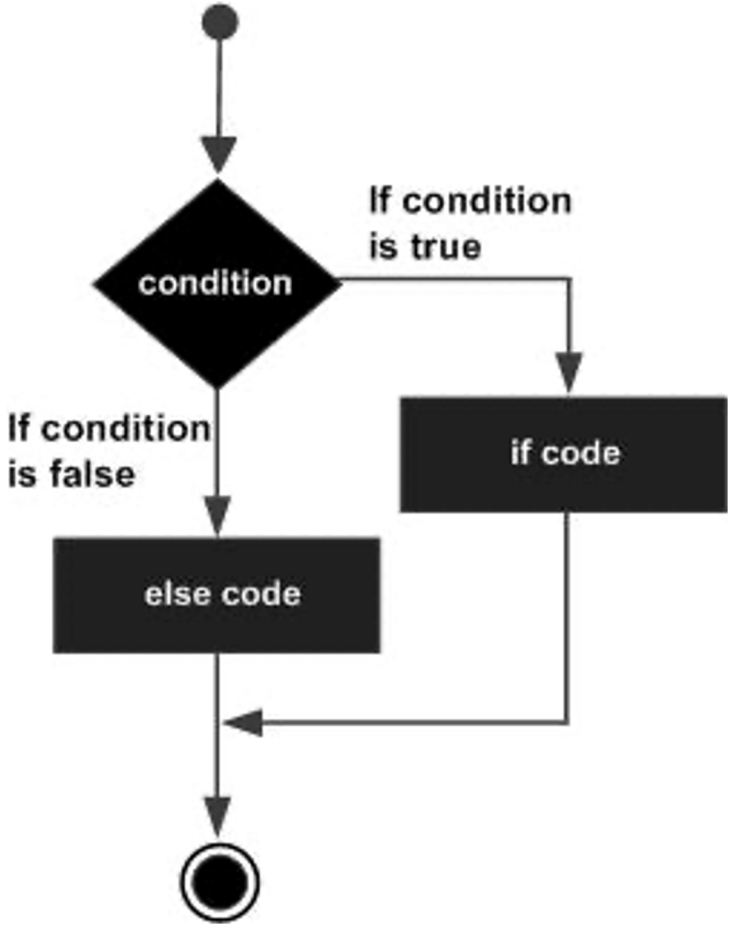
Functions

```
>>> def sum(a,b):
...     return a+b
...
>>> sum(1,2)
3
>>> def sum_with_defaults(a = 1, b = 10):
...     return a+b
...
>>> sum_with_defaults()
11
>>> sum_with_defaults(5)
15
```

Functions can also be called with keyword arguments `kwarg=value` (Google it)

Control Flow Statements: if / elif / else

Conditionally execute statements/blocks. The keyword ‘elif’ is short for ‘else if’, and is useful to avoid excessive indentation. An if … elif … elif … sequence is a substitute for the switch or case statements found in other languages.

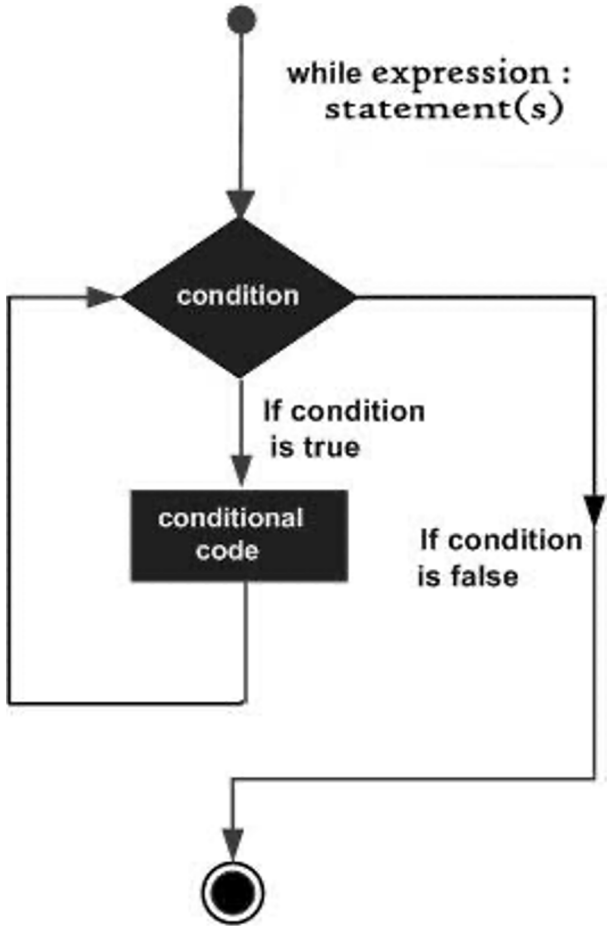


if/elif/else

```
>>> x = 0
>>> if x < 10:
...     print("Less than 10")
... elif x > 10 and x < 20:
...     print("Between 10 and 20")
... else:
...     print("More than 20")
...
Less than 10
```

Control Flow Statements: while

While is used for repeated execution *as long as an expression is true.*

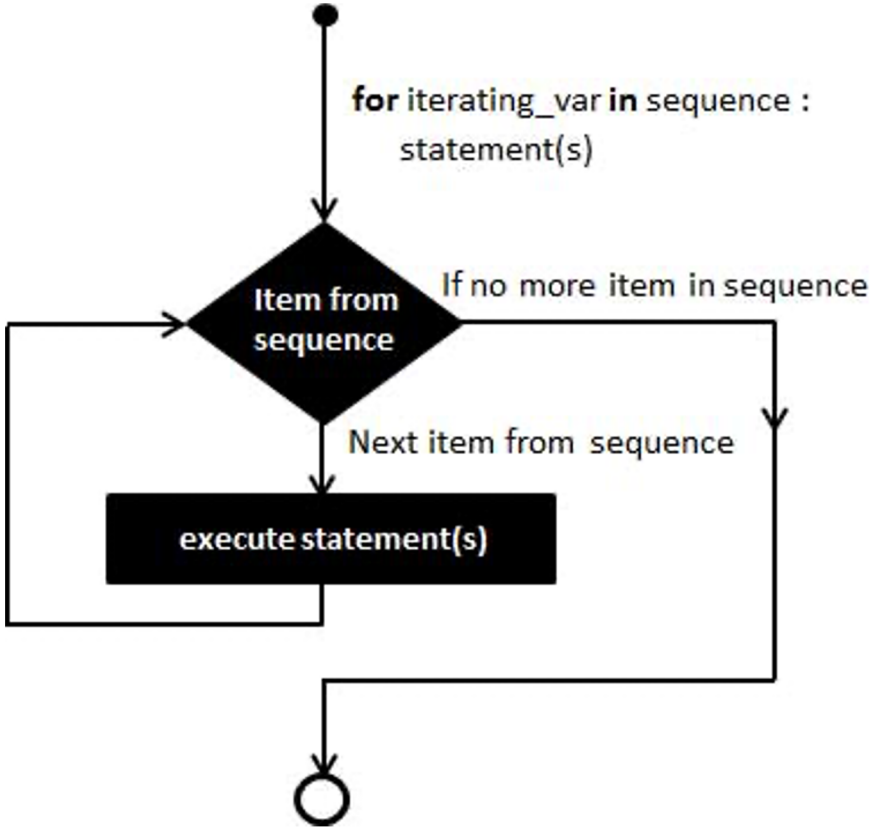


while

```
>>> x = 0
>>> while x < 2:
...     print(x)
...     x += 1
...
0
1
```

Control Flow Statements: for

While is used *to iterate over the elements of a sequence* (such as a string, tuple or list) or other iterable object



for

```
>>> for i in [0, 1, 2]:
...     print(i)
...     i=5
...
0
1
2
>>> # Note that i=5 has no effect
```

Control Flow Statements: break and continue

Break is used to *stop the execution of the loop.* It breaks out of the innermost enclosing for or while loop. **Continue** *continues with the next iteration* of the loop.

break

```
>>> for n in range(2,8): # range(2,8) == [2, 3, 4, 5, 6, 7]
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else: # else can also be used in this context (note intendation)
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
```

Control Flow Statements: break and continue

Break is used to *stop the execution of the loop.* It breaks out of the innermost enclosing for or while loop. **Continue** *continues with the next iteration* of the loop.

continue

```
>>> for num in range(2, 10): # range(2,10) == [2, 3, 4, 5, 6, 7, 8, 9]
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

Exception handling : try and except

Concept: for code within a **try** block, when an error occurs, an **exception is raised**, and the program execution is suspended.

- If the exception is not caught, the program terminates
- If it is, using **except**, the program resumes its execution in the exception handler
  - Usually one except block per exception type (multiple can occur)

Exceptions

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
```

An exception is raised using, e.g., `raise ValueError("an exception")`

Exception handling : try and except

Some common exceptions are (more at https://docs.python.org/3/library/exceptions.html)

- SyntaxError: syntax error
- ValueError: when an argument that has the right type but an inappropriate value
- OSError: when a system function returns a system-related error
- KeyError: when a mapping (dictionary) key is not found in the set of existing keys

User Defined Exceptions

```
>>> class MyException(Exception):
...     pass
...
>>> raise MyException("my message") # More on classes later
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyException: my message
```

Using Exceptions

Using Exceptions

```
>>> try:
...     if error1: # Some error condition
...         raise MyException("A problem occurred.")
...     if error2: # Some error condition
...         raise ValueError("Wrong value provided.")
... except MyException as e:
...     print(e)
... except ValueError as e:
...     print(e)
...
```

Helper functions

Python has a lot of functions that can be used to perform lots of tasks. The "standard library" is quite extensive, and there are also lots of packages.

range

```
>>> list(range(0,5)) # Note that 4 is excluded from the range
[0, 1, 2, 3, 4]
>>> list(range(0,5,2)) # With a step of 2
[0, 2, 4]
>>> [x for x in range(0,5,2)] # Yes, this is valid in python
[0, 2, 4]
```

Let’s try some of this

Importing a Module

- A module is a collection of classes, functions, etc...
- Modules can be shared between multiple applications
- Modules are a **very important aspect and allow for code re-use**

Importing modules

```
>>> import module
>>> import module.submodule
>>> import module as m
>>> from module import submodule
```

PIP :: Package Installer for Python

- PIP is a package manager which allows you to conveniently install packages / modules. More info here.

Installing pip

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ python3 get-pip.py
```

Using pip

```
$ pip3 install --user numpy
Collecting numpy
  Downloading numpy-1.18.2-cp37-cp37m-macosx_10_9_x86_64.whl
(15.1 MB)
     |████████████████████████████████| 15.1 MB 126 kB/s
Installing collected packages: numpy
Successfully installed numpy-1.18.2
```

Creating a Module

- One can create his/her own modules by grouping functions/classes in one or multiple files in a folder

Importing modules

```
$ mkdir MyModule
$ touch MyModule/__init__.py
$ touch MyModule/foo.py
$ touch MyModule/bar.py
# Edit the files...
$ python3
>>> import MyModule
>>> import MyModule.foo
>>> import MyModule.bar
```

- A module needs to be in the current directory or accessible via the PYTHONPATH environment variable

```
export PYTHONPATH=/path/of/my/module1:/path/of/my/module2
```

An exercise

Exercise

Write a command-line tool to append a line of text (read from the keyboard) to a file. Don't go past the ANSWER section !!

Expected output

```
$ ./append_to_file.py
Hello
$ cat myFile.txt
Hello
```

Tips:

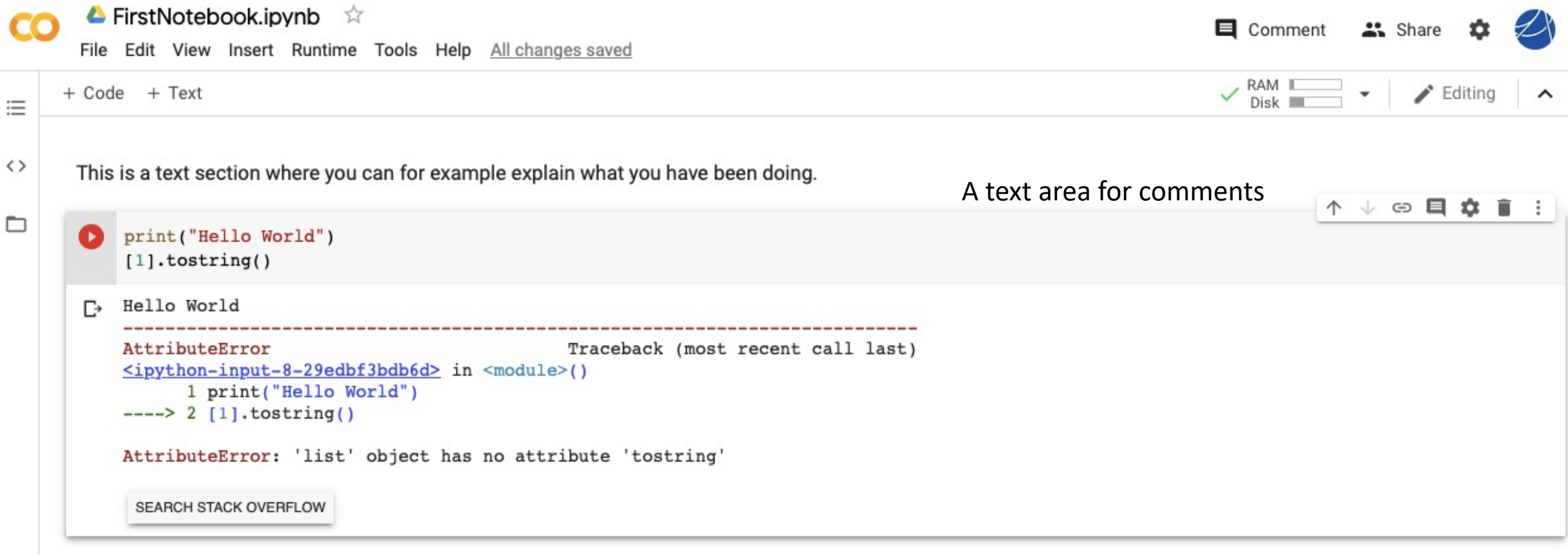
- The module sys should be called using `import sys` and you can use `sys.stdin` as a file to read a line from your terminal
- In order to call the script directly, don't forget the shebang `#!/usr/bin/env python3`
  and to make the file executable (using `chmod +x`)

Python (Jupyter) Notebooks

- Jupyter Notebook documents are both **human-readable documents** containing the **analysis description and the results** (figures, tables, etc..) as well as **executable documents which can be run to perform data analysis**. See, e.g.: https://jupyter-notebook-beginner-guide.readthedocs.io

- **Google Colaboratory (https://colab.research.google.com/)**
  - Provides convenient access to Python Notebooks, which can be shared and worked on collaboratively
  - There are of course other providers, some even free
  - See : https://colab.research.google.com/notebooks/intro.ipynb

Google Colaboratory



A nice display for errors stack / help to search on Stack Overflow

Notebooks

- A fancier example with plots embedded in the notebook
- This is very useful to resume work and get a stable environment
- In the back-end, there is a python kernel that remembers what you ran in previous code blocks in the document

FirstNotebook.ipynb

File Edit View Insert Runtime Tools Help All cha

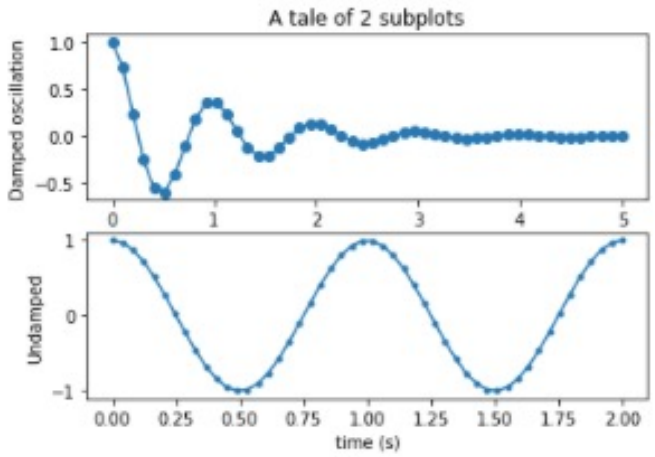
Comment Share

+ Code + Text

RAM Disk

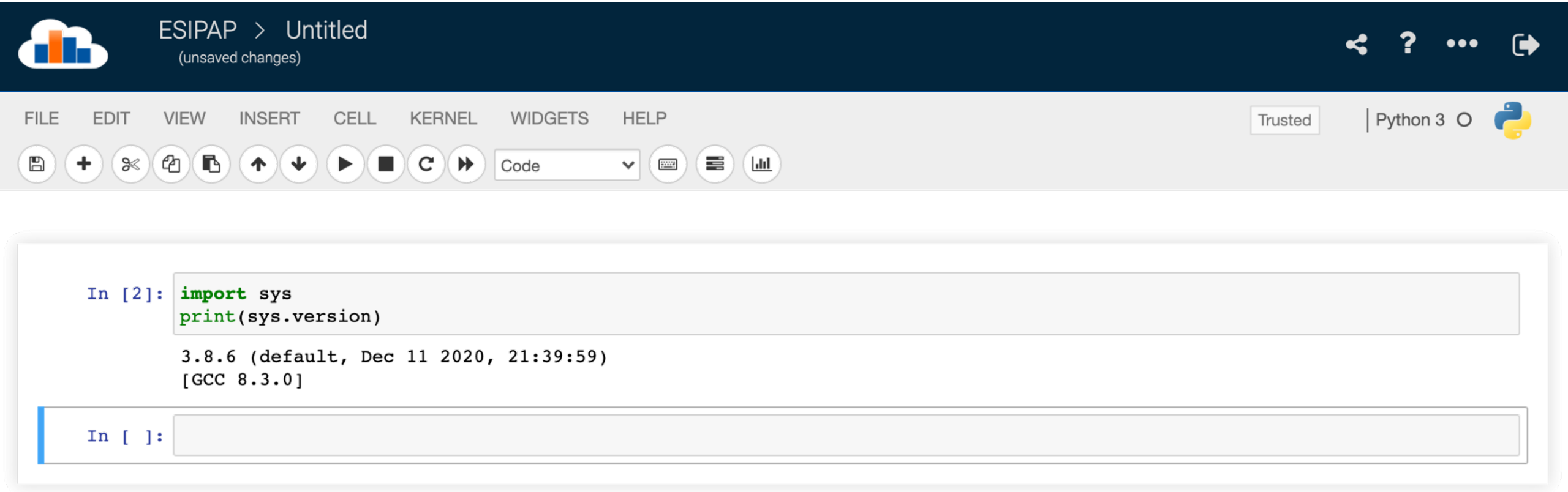
Editing

```
import numpy as np
import matplotlib.pyplot as plt


x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'o-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, '.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')

plt.show()
```



CERN SWAN

- If you have a CERN account, use SWAN: https://swan.cern.ch
- (Conveniently) connects to your CERNbox/EOSuser space for storage
- Support for multiple languages (also C++) and links with ROOT



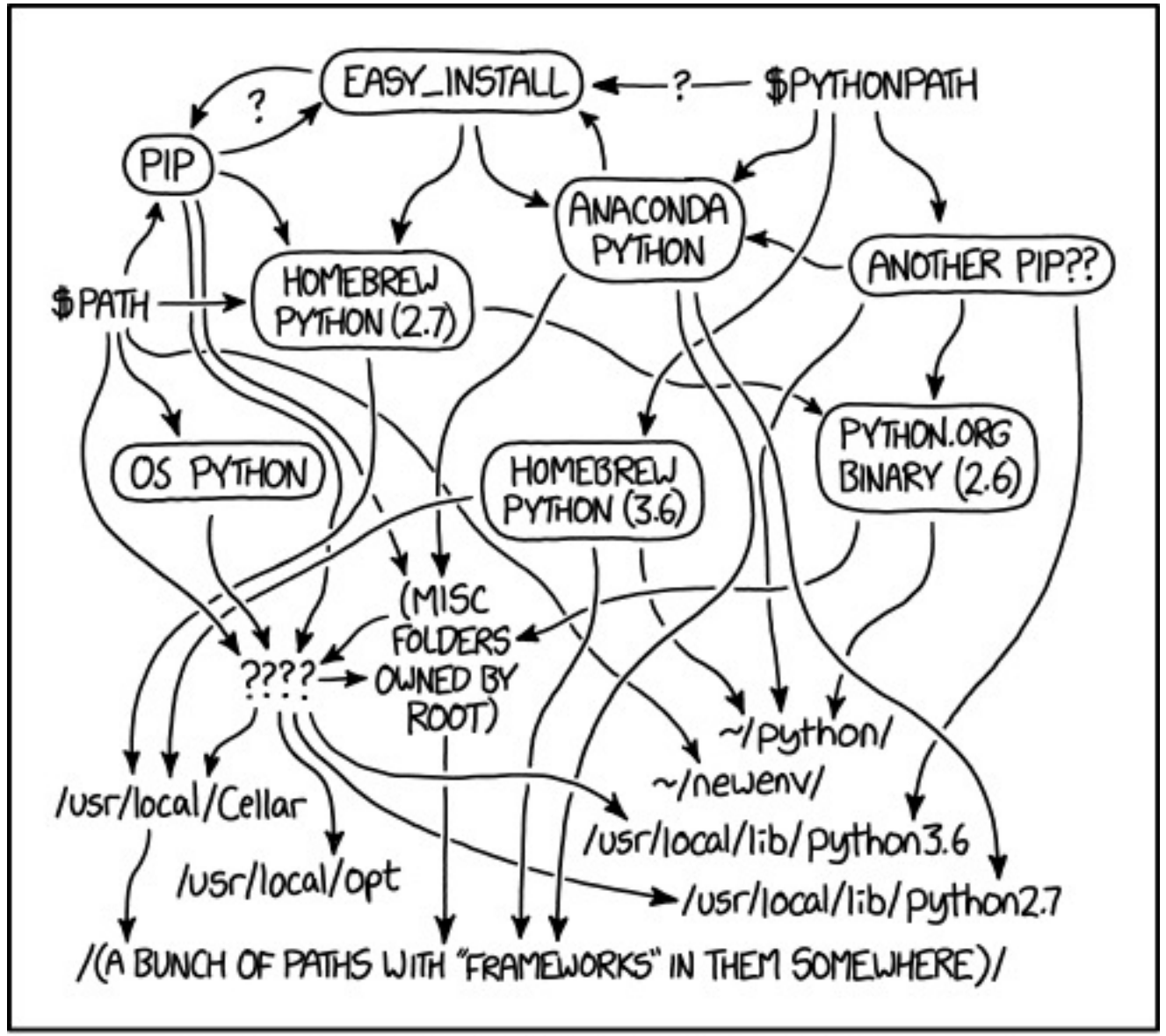
Locally

- Make sure you run python3 (on ”older” systems, python is still v2)
  - Ideally, you should have a version >= 3.6

Using Jupyter locally

```
$ python3 --version
Python 3.8.2
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ pip install jupyter
...
(venv) $ ./venv/bin/jupyter notebook
```

- This will open a web browser for you, starting from your local folder, where you can create new notebooks and edit existing ones



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

https://xkcd.com/1987/

A word on virtualenv

- Python applications will often use packages and modules that don't come as part of the standard library.
- Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the application may be written using an obsolete version of the library's interface.
- This means it may not be possible for one Python installation to meet the requirements of every application. [a.k.a. dependency hell]
  - If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run.
- More info here: https://docs.python.org/3/tutorial/venv.html

virtualenv

```
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $
```

Exercise (if you have no python on your system)

Using some python notebook, e.g. using Google Colaboratory, SWAN, or local resources, write a tool to append a line of text (provided in-code) to a file.

Don't go past the ANSWER section !!

Tips:

- You can use the question mark (!) to escape to the shell and run (some) system commands, i.e.
  `!cat myFile.txt`
  will show you the content of the file called myFile.txt

Answers

Please try the exercise first…

A possible solution

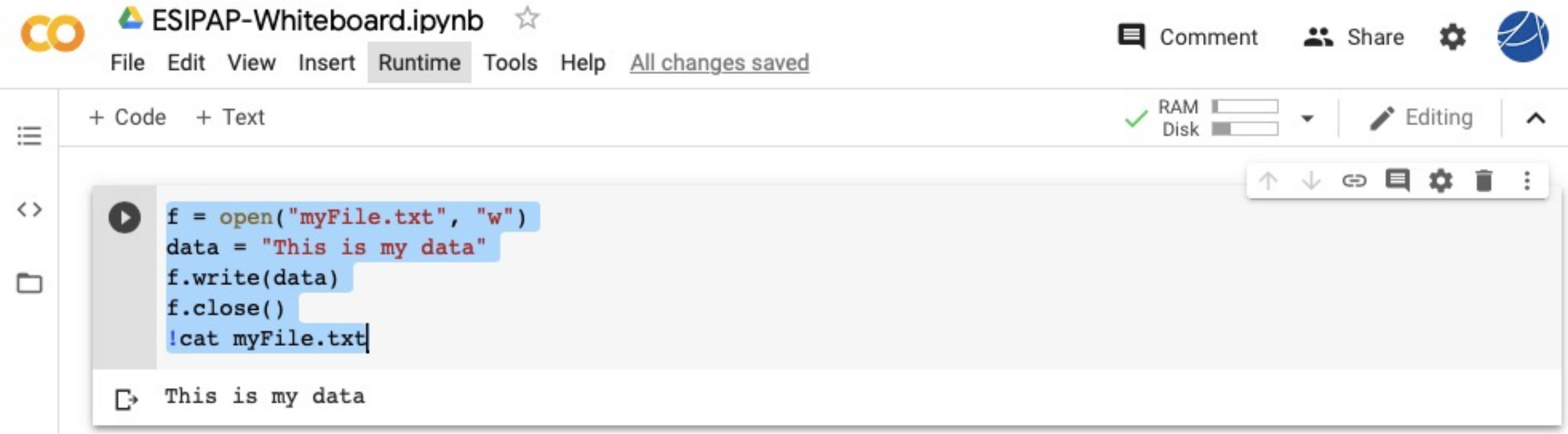
append_to_file.py

```
#!/usr/bin/env python3

import sys

f = open("myFile.txt", "w")
data = sys.stdin.readline()
f.write(data)
f.close()
```

A possible solution in Google Colaboratory

In your Notebook

```
f = open("myFile.txt", "w")
data = "This is my data"
f.write(data)
f.close()
!cat myFile.txt
```



Objects in Python

Object Oriented Programming (OOP)

- OOP refers to a type of computer programming in which programmers define the **data type of a data structure** and the **types of operations** (**methods**) that can be applied to the data structure
- Classes provide a means of **bundling data and functionality together**
- Creating a new class creates a **new type of object**, allowing **new instances** of that type to be made
- Each instance can have **attributes** attached to it for **maintaining its state**
- Instances can also have methods (defined by its class) to **modify its state**
- The **class inheritance** mechanism allows multiple **base** classes, a **derived** class can **override any methods of its base class** or classes, and a **method can call the method of a base class with the same name**.

Some definitions

- **Class**: A user-defined prototype for an **object** that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via **dot notation**.
- **Class variable**: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables aren't used as frequently as instance variables are.
- **Instance variable**: A variable that is defined inside a method and belongs only to the current instance of a class.
- **Method**: A special kind of function that is defined in a class definition.
- **Instance**: An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation**: The creation of an instance of a class.

Class and Instance Variables

Python Classes

```
class Dog:
    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

Class Inheritance

Class Inheritance

```
Class Base1(object): # <class 'object'> is the root of all classes
    <statement-1>
    .
    <statement-N>

class DerivedClassName(Base1):
    <statement-1>
    .
    <statement-N>

Class Base2(object):
    <statement-1>
    .
    <statement-N>

class DerivedClassName(Base1, Base2):  # Multiple inheritance
    <statement-1>
    .
    <statement-N>
```

Class Inheritance

Class Inheritance

```
class MyClass1(object):
    def __init__(self, foo):
        self.foo = foo
    def print(self):
        print(self.foo)
    def hello(self):
        print('hello %s' % self.foo)

class MyClass2(MyClass1):
    def __init__(self, foo, bar):
        super().__init__(foo)     # call the parent constructor
        self.bar = bar
    def print(self):              # the method is overridden
        super().print()           # call the parent method
        print(self.bar)
```

Class Inheritance

Class Inheritance

```
>>> # Previous slide saved as MyModule.py in current folder
>>> import MyModule
>>> x = MyModule.MyClass1("Hello")
>>> y = MyModule.MyClass2("One", "Two")
>>> x.print()
Hello
>>> y.print()
One
Two
```

Class Properties

Class Properties

```
class Celsius:
    def __init__(self, temperature = 0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    def get_temperature(self):
        print("Getting value")
        return self._temperature

    def set_temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        print("Setting value")
        self._temperature = value

    temperature = property(get_temperature,set_temperature)
```

Class Properties

Class Properties

```
>>> from Celsius import Celsius # To avoid typing Celsius.Celsius
>>> x = Celsius()
Setting value
>>> x.set_temperature(10)
Setting value
>>> x.get_temperature()
Getting value
10
>>> x.to_fahrenheit()
Getting value
50.0
>>> x.set_temperature(-500)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/karolos/CERNbox/ESIPAP/Python/Celsius.py", line 14, in
set_temperature
    raise ValueError("Temperature below -273 is not possible")
ValueError: Temperature below -273 is not possible
```

Operator Overloading

complex.py

```
class complex:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __add__(self, other):
        return self.a + other.a, self.b + other.b

    def __str__(self):
        return self.a, self.b
```

Operator Overloading

```
>>> from complex import complex
>>> Ob1 = complex(1, 2)
>>> Ob2 = complex(2, 3)
>>> Ob3 = Ob1 + Ob2
>>> print(Ob3)
(3, 5)
>>>
```

Another exercise

Exercise

Write a vector class (arbitrary dimension)

- **Create the module Vector**
- **Create the class Vector**
- **Write the constructor `[def __init__(self, dim)]`**
- **Overload operators**

Use the internet for help, but don't google vector class for an answer

Useful Python Libraries

import os

Miscellaneous operating system interfaces

- More info: https://docs.python.org/3/library/os.html

Lots of functions of the POSIX standard:

- mkdir, rmdir, remove, chmod, etc.
- environ[], setenv(), getenv()
- system(), popen() … to execute shell commands

- `import os.path` for path manipulations (exists, is_dir, etc.)
  - See https://docs.python.org/3/library/os.path.html

Other (System) Libraries

- `import glob`: file wildcards
- `import re`: regular expressions
- `import math`: mathematical functions
- `import random`: random number generation
- `import urllib`: fetching resources from the internet
- `import time, datetime`: time manipulation
- `import zlib`: compression

Libraries Provided by 3rd Parties

- There is a very broad ecosystem of Python libraries provided by third parties. Here we just name a few.

- SciPy: Python-based ecosystem of open-source software for mathematics, science, and engineering. See https://www.scipy.org
  - NumPy: base for N-dimensional array package
  - SciPy: fundamental library for scientific computing
  - Matplotlib: for 2D/3D plotting
  - IPyton: enhanced interactive console
  - Sympy: symbolic mathematics
  - Pandas: Data structure and analysis

Numpy :: pip install numpy

- Numpy: the fundamental package for scientific computing with Python
- Powerful N-dimensional arrays: fast and versatile, the NumPy vectorization, indexing, and broadcasting concepts are de-facto standards
- Numerical computing tools: comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more
- Interoperable: supports a wide range of hardware and computing platforms, and plays well with distributed, GPU, and sparse array libraries
- Performant: well-optimized C code. Enjoy the flexibility of Python with the speed of compiled code.
- Easy to use: high level syntax makes it accessible and productive for programmers from any background or experience level
- Open source: BSD license

Numpy arrays

Numpy

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64’
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<class 'numpy.ndarray’>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<class 'numpy.ndarray'>
```

Numpy arrays : a note on syntax

Numpy

```
>>> a = np.array(1,2,3,4)    # WRONG
Traceback (most recent call last):
  ...
TypeError: array() takes from 1 to 2 positional arguments but 4 were
given
>>> a = np.array([1,2,3,4])  # RIGHT
```

Numpy

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

Numpy array initialisation

Numpy

```
>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.ones( (2,3,4), dtype=np.int16 )                # dtype can also be
specified
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
>>> np.empty( (2,3) )                                 # uninitialized
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],  # may vary
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

Numpy

Numpy

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
>>> from numpy import pi
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
>>> x = np.linspace( 0, 2*pi, 100 )        # useful to evaluate function at lots
of points
>>> f = np.sin(x)
```

Numpy

Numpy

```
>>> a = np.arange(6)                         # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>>
>>> b = np.arange(12).reshape(4,3)           # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
>>> c = np.arange(24).reshape(2,3,4)         # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Numpy : smart printing

Numpy

```
>>> print(np.arange(10000))
[   0    1    2 ... 9997 9998 9999]
>>>
>>> print(np.arange(10000).reshape(100,100))
[[   0    1    2 ...   97   98   99]
 [ 100  101  102 ...  197  198  199]
 [ 200  201  202 ...  297  298  299]
 ...
 [9700 9701 9702 ... 9797 9798 9799]
 [9800 9801 9802 ... 9897 9898 9899]
 [9900 9901 9902 ... 9997 9998 9999]]
```

No time to cover everything … check https://numpy.org

Pandas :: pip install pandas

- A fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language
- Used very often, and (IMO) a must-know
- Locally you install them using pip

Importing Pandas

```
(venv) $ pip install pandas numpy
$ python3
...
>>> # Customary imports for Numpy and Pandas
>>> import numpy as np
>>> import pandas as pd
```

- Fundamental data structures of Pandas are **Series** and **DataFrame**

Pandas Series

Series

```
>>> s = pd.Series(np.random.randn(4), index=["a", "b", "c", "d"])
>>> s
a    0.959893
b    0.176460
c   -0.433848
d   -0.469920
dtype: float64
>>> s.index
Index(['a', 'b', 'c', 'd'], dtype='object')
>>> pd.Series(np.random.randn(5))
0    0.669462
1   -0.599999
2    0.956523
3   -0.300907
4    2.535160
dtype: float64
```

Series is a 1D labelled array capable of holding any data type (integers, strings, floating point numbers, objects, etc.). The axis labels are collectively referred to as the index.

Pandas Series from a dictionary or a scalar value

Series

```
>>> d = {"b": 1, "a": 0, "c": 2}
>>> pd.Series(d)
b    1
a    0
c    2
dtype: int64
>>> pd.Series(d, index=["b", "c", "d", "a"])
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
>>> pd.Series(5.0, index=["a", "b", "c", "d", "e"])
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Using Pandas Series

Series

```
>>> s[0]
0.9598926249543338
>>> s[:3]
a    0.959893
b    0.176460
c   -0.433848
dtype: float64
>>> s[s > s.median()]
a    0.959893
b    0.176460
dtype: float64
>>> s[[3,2]]
d   -0.469920
c   -0.433848
dtype: float64
>>> np.exp(s)
a    2.611416
b    1.192987
c    0.648011
d    0.625052
dtype: float64
```

Using Pandas Series

Series

```
>>> s.dtype
dtype('float64')
>>> s.array
<PandasArray>
[ 0.9598926249543338, 0.17646021129740513, -0.433847601312536 1,
 -0.4699200545656808]
Length: 4, dtype: float64
>>> s.index
Index(['a', 'b', 'c', 'd'], dtype='object')
>>> s.to_numpy()
array([ 0.95989262,  0.17646021, -0.4338476 , -0.46992005])
```

Using Pandas Series

Series

```
>>> s["a"]
0.9598926249543338
>>> s["d"] = 12.0
>>> s
a     0.959893
b     0.176460
c    -0.433848
d    12.000000
dtype: float64
>>> "d" in s
True
>>> "e" in s
False
>>> s["f"]
Traceback (most recent call last):
...
KeyError: 'f'
>>> s.get("f")
>>> s.get("f", np.nan)
nan
```

Using Pandas Series

Series

```
>>> s+s
a     1.919785
b     0.352920
c    -0.867695
d    24.000000
dtype: float64
>>> s*2
a     1.919785
b     0.352920
c    -0.867695
d    24.000000
dtype: float64
>>> s[1:] + s[:-1] # Operations done correctly based on index label
a          NaN
b     0.352920
c    -0.867695
d          NaN
dtype: float64
```

Pandas DataFrame

DataFrame

```
>>> d = {
...      "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"]),
...      "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c",
"d"]),
... }
>>> df = pd.DataFrame(d)
>>> df
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0
```

DataFrame is a 2D labelled data structure with columns of potentially different types.

Pandas DataFrame from list

DataFrame

```
>>> d = {
...      "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"]),
...      "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c",
"d"]),
... }
>>> pd.DataFrame(d, index=["d", "b", "a"])
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0
>>> pd.DataFrame(d, index=["d", "b", "a"], columns=["two", "three"])
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
>>> df.index
Index(['a', 'b', 'c', 'd'], dtype='object')
>>> df.columns
Index(['one', 'two'], dtype='object')
```

Pandas DataFrame

DataFrame

```
>>> d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0,
>>> pd.DataFrame(d)
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0
>>> pd.DataFrame(d, index=["a", "b", "c", "d"])
   one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```

Pandas DataFrame from a structured array

DataFrame

```
>>> np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "a10")])
array([(0, 0., b''), (0, 0., b'')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])
>>> data = np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "a10")])
>>> data[:] = [(1, 2.0, "Hello"), (2, 3.0, "World")]
>>> pd.DataFrame(data)
   A    B         C
0  1  2.0  b'Hello'
1  2  3.0  b'World'
>>> pd.DataFrame(data, index=["first", "second"])
        A    B         C
first   1  2.0  b'Hello'
second  2  3.0  b'World'
>>> pd.DataFrame(data, columns=["C", "A", "B"])
           C  A    B
0  b'Hello'  1  2.0
1  b'World'  2  3.0
```

Pandas DataFrame from list of dicts

DataFrame

```
>>> data2 = [{"a": 1, "b": 2}, {"a": 5, "b": 10, "c": 20}]
>>> pd.DataFrame(data2)
   a   b     c
0  1   2   NaN
1  5  10  20.0
>>> pd.DataFrame(data2, index=["first", "second"])
        a   b     c
first   1   2   NaN
second  5  10  20.0
>>> pd.DataFrame(data2, columns=["a", "b"])
   a   b
0  1   2
1  5  10
```

Pandas DataFrame from dict of tuples

DataFrame

```
>>> pd.DataFrame(
...     {
...         ("a", "b"): {("A", "B"): 1, ("A", "C"): 2},
...         ("a", "a"): {("A", "C"): 3, ("A", "B"): 4},
...         ("a", "c"): {("A", "B"): 5, ("A", "C"): 6},
...     }
...  )
     a
     b  a  c
A B  1  4  5
  C  2  3  6
```

Creates a **multiindexed** frame

Operations on DataFrame

DataFrame

```
>>> df["one"]
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
>>> df["three"] = df["one"] * df["two"]
>>> df["flag"] = df["one"] > 2
>>> df
   one  two  three   flag
a  1.0  1.0    1.0  False
b  2.0  2.0    4.0  False
c  3.0  3.0    9.0   True
d  NaN  4.0    NaN  False
```

Operations on DataFrame

DataFrame

```
>>> del df["two"]
>>> three = df.pop("three")
>>> df
   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
>>> df["foo"] = "bar"
>>> df
   one   flag  foo
a  1.0  False  bar
b  2.0  False  bar
c  3.0   True  bar
d  NaN  False  bar
```

Operations on DataFrame

DataFrame

```
>>> df
   one   flag  foo
a  1.0  False  bar
b  2.0  False  bar
c  3.0   True  bar
d  NaN  False  bar
>>> df["one_trunc"] = df["one"][:2]
>>> df
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
```

Operations on DataFrame

DataFrame

```
>>> df
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
>>> df.insert(1, "bar", df["one"])
>>> df
   one  bar   flag  foo  one_trunc
a  1.0  1.0  False  bar        1.0
b  2.0  2.0  False  bar        2.0
c  3.0  3.0   True  bar        NaN
d  NaN  NaN  False  bar        NaN
```

Reading in data from a CSV file

- DataFrames are really powerful at reading data from various sources, such as CSV (comma-separated values) files

DataFrame

```
>>> import os
>>> os.system("wget https://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data >& /dev/null")
>>> iris = pd.read_csv("iris.data", names=["SepalLength", "SepalWidth",
"PetalLength", "PetalWidth", "Name"])
>>> iris.head()
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
3          4.6         3.1          1.5         0.2  Iris-setosa
4          5.0         3.6          1.4         0.2  Iris-setosa
```

Operations on DataFrame

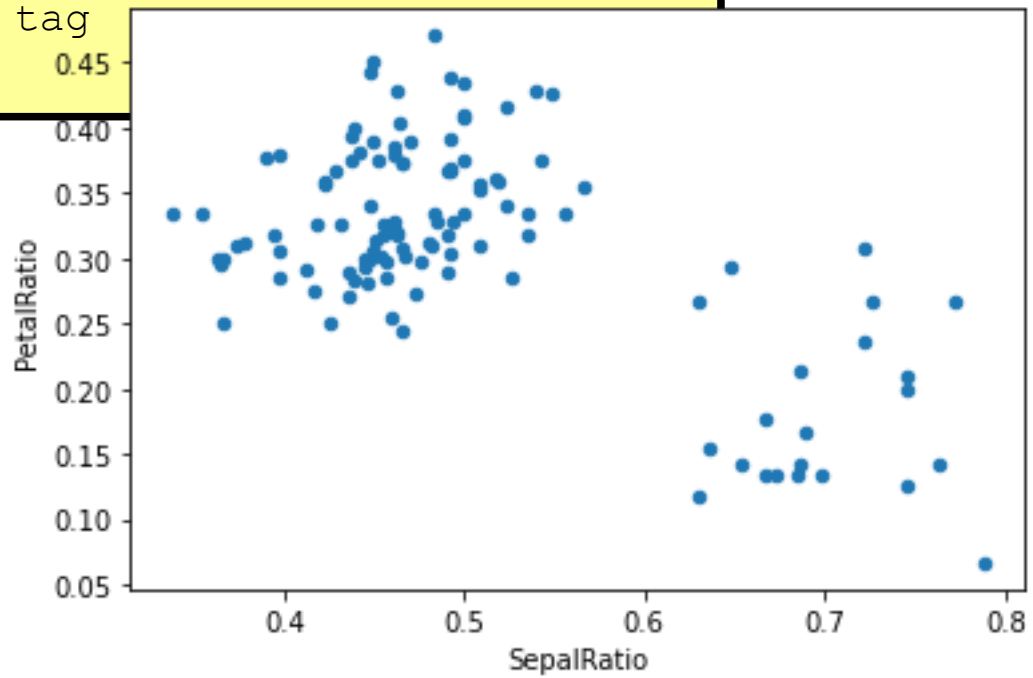
DataFrame

```
>>> iris.columns
Index(['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth', 'Name'],
dtype='object')
>>> iris.assign(sepal_ratio=iris["SepalWidth"] / iris["SepalLength"]).head()
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
>>> iris.assign(sepal_ratio=lambda x: (x["SepalWidth"] / x["SepalLength"])).head()
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
>>> iris.head(2)
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
```

Operations on DataFrame

DataFrame

```
>>> import matplotlib
>>> (
...       iris.query("SepalLength > 5")
...       .assign(
...           SepalRatio=lambda x: x.SepalWidth / x.SepalLength,
...           PetalRatio=lambda x: x.PetalWidth / x.PetalLength,
...       )
...       .plot(kind="scatter", x="SepalRatio", y="PetalRatio")
... )
Unable to revert mtime: /Library/Fonts
Fontconfig warning: ignoring UTF-8: not a valid region tag
<AxesSubplot:xlabel='SepalRatio', ylabel='PetalRatio'>
```



More on this dataset later …

Operations on DataFrame

DataFrame

```
>>> dates = pd.date_range("20130101", periods=6)
>>> dates
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
>>> df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))
>>> df
                   A         B         C         D
2013-01-01 -0.616776  2.255949 -0.804261  0.136389
2013-01-02  2.036988  0.133639 -2.194401 -1.314238
2013-01-03 -1.955609  0.814910 -0.450114 -2.416581
2013-01-04 -1.496272 -2.286047 -0.255013  0.302078
2013-01-05  0.527517  0.704457 -0.248928  0.040991
2013-01-06  1.990637  0.713619  0.499214  1.528811
>>> df.tail(3)
                   A         B         C         D
2013-01-04 -1.496272 -2.286047 -0.255013  0.302078
2013-01-05  0.527517  0.704457 -0.248928  0.040991
2013-01-06  1.990637  0.713619  0.499214  1.528811
```

Too many things to do with DataFrame …

Operations on DataFrame

DataFrame

```
>>> df
                   A         B         C         D
2013-01-01 -0.616776  2.255949 -0.804261  0.136389
2013-01-02  2.036988  0.133639 -2.194401 -1.314238
2013-01-03 -1.955609  0.814910 -0.450114 -2.416581
2013-01-04 -1.496272 -2.286047 -0.255013  0.302078
2013-01-05  0.527517  0.704457 -0.248928  0.040991
2013-01-06  1.990637  0.713619  0.499214  1.528811
>>> df.iloc[3]
A   -1.496272
B   -2.286047
C   -0.255013
D    0.302078
Name: 2013-01-04 00:00:00, dtype: float64
>>> df.iloc[3:5, 0:2]
                   A         B
2013-01-04 -1.496272 -2.286047
2013-01-05  0.527517  0.704457
>>> df[df["A"] > 0]
                   A         B         C         D
2013-01-02  2.036988  0.133639 -2.194401 -1.314238
2013-01-05  0.527517  0.704457 -0.248928  0.040991
2013-01-06  1.990637  0.713619  0.499214  1.528811
```

A word on missing data

DataFrame

```
>>> df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ["E"])
>>> df1.loc[dates[0] : dates[1], "E"] = 1
>>> df1
                   A         B         C         D    E
2013-01-01 -0.616776  2.255949 -0.804261  0.136389  1.0
2013-01-02  2.036988  0.133639 -2.194401 -1.314238  1.0
2013-01-03 -1.955609  0.814910 -0.450114 -2.416581  NaN
2013-01-04 -1.496272 -2.286047 -0.255013  0.302078  NaN
>>> df1.dropna(how="any")
                   A         B         C         D    E
2013-01-01 -0.616776  2.255949 -0.804261  0.136389  1.0
2013-01-02  2.036988  0.133639 -2.194401 -1.314238  1.0
>>> df1.fillna(value=5)
                   A         B         C         D    E
2013-01-01 -0.616776  2.255949 -0.804261  0.136389  1.0
2013-01-02  2.036988  0.133639 -2.194401 -1.314238  1.0
2013-01-03 -1.955609  0.814910 -0.450114 -2.416581  5.0
2013-01-04 -1.496272 -2.286047 -0.255013  0.302078  5.0
>>> pd.isna(df1).tail(2)
                A      B      C      D     E
2013-01-03  False  False  False  False  True
2013-01-04  False  False  False  False  True
```

A word on missing data

- As data comes in many shapes and forms, pandas aims to be flexible with regard to handling missing data.
- While NaN is the default missing value marker for reasons of computational speed and convenience, we need to be able to easily detect this value with data of different types: floating point, integer, boolean, and general object.
- In many cases, however, the Python None will arise and we wish to also consider that “missing” or “not available” or “NA”.
- To consider inf and -inf as “NA”, need to set `pandas.options.mode.use_inf_as_na = True`
- Note: Starting from pandas 1.0, an experimental `pd.NA` value (singleton) is available to represent scalar missing values.

Stats in Pandas (excluding missing data)

DataFrame

```
>>> df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))
>>> df
                   A         B         C         D
2013-01-01 -0.874710 -0.479047  0.349441 -0.069385
2013-01-02  0.815037  2.159031 -0.230224  0.001993
2013-01-03  0.393732 -0.418193  0.638911 -0.467236
2013-01-04 -0.744395 -1.551088  0.622943  0.622890
2013-01-05  0.239997 -0.971747 -1.119259 -0.512740
2013-01-06 -0.723792  0.131198  0.992990  0.421122
>>> df.mean()
A   -0.149022
B   -0.188308
C    0.209134
D   -0.000559
dtype: float64
>>> df.mean(1)
2013-01-01   -0.268425
2013-01-02    0.686459
2013-01-03    0.036804
2013-01-04   -0.262413
2013-01-05   -0.590937
2013-01-06    0.205380
Freq: D, dtype: float64
```

Grouping

DataFrame

```
>>> df["E"] = "Good"
>>> df.loc[dates[3]:dates[5], "E"] = "Bad"
>>> df
                   A         B         C         D     E
2013-01-01 -0.874710 -0.479047  0.349441 -0.069385  Good
2013-01-02  0.815037  2.159031 -0.230224  0.001993  Good
2013-01-03  0.393732 -0.418193  0.638911 -0.467236  Good
2013-01-04 -0.744395 -1.551088  0.622943  0.622890   Bad
2013-01-05  0.239997 -0.971747 -1.119259 -0.512740   Bad
2013-01-06 -0.723792  0.131198  0.992990  0.421122   Bad
>>> df.groupby("E").sum()
            A         B         C         D
E
Bad  -1.228191 -2.391636  0.496674  0.531271
Good  0.334058  1.261791  0.758128 -0.534628
>>> df.groupby("E").mean()
            A         B         C         D
E
Bad  -0.409397 -0.797212  0.165558  0.177090
Good  0.111353  0.420597  0.252709 -0.178209
```

One could keep going for a very long time … check doc @ https://pandas.pydata.org

Libraries Provided by 3rd Parties (2)

- Of course, there are also the machine learning libraries…

- **Tensorflow:** TensorFlow is an end-to-end python machine learning library for performing high-end numerical computations: can handle deep neural networks for image recognition, handwritten digit classification, recurrent neural networks, NLP (Natural Languae Processing), word embedding, etc.
- **Keras:** leading open-source Python library written for constructing neural networks and machine learning projects.
- **Scikit-learn:** another prominent open-source Python machine learning library with a broad range of clustering, regression and classification algorithms.
- **PyTorch:** deep neural networks and Tensor computation with GPU acceleration are the two high-end features of the PyTorch
- **Theano:** aims to boost development time and execution time of ML apps, particularity in deep learning algorithms. (Syntax is not beginner-friendly.)

A few examples with Scikit Learn and Keras/TF

- In the tutorial, we'll cover some basic examples from machine learning (ML) and deep learning (DL) though these are beyond the scope of this course on Python.
- Yet, they allow you to understand how Python can be used in a production environment.
- Note that these examples come from an ML/DL course, and I don't have time to cover the basics here, but you can read up online.
- Examples (in GitLab): https://gitlab.cern.ch/karolos-potamianos/esipap/Python-Exercises
  - Iris-ML-Example.ipynb: Using ML to classify plants (Iris)
  - Iris.ipynb : using a neural network to classify plants (Iris)
  - MNIST-TF-Keras-CNN.ipynb : Convolutional Neural Networks with MNIST dataset
  - MNIST-TF-Keras-NN-Basic.ipynb: Neural Networks with MNIST dataset

Quantum Computing Simulation

- Quantum computing offer exciting perspectives for the future (of course, provided that they can deliver), but the hardware is not yet widely available. Until then, we have to rely on **simulation** to try them out.

Two very popular Python frameworks:

- PennyLane (https://pennylane.ai) is a cross-platform Python library for differentiable programming of quantum computers.
  - Essentially allows training a quantum computer the same way as a neural network.
- Tensorflow Quantum (https://www.tensorflow.org/quantum) is a library for hybrid quantum-classical machine learning

- Unfortunately I don't have time to discuss these (they go well beyond this class) but you may have some fun exploring this

Conclusion

- Python is an interpreted, high-level, general-purpose programming language.
- Python's design philosophy emphasizes code readability with its notable use of significant whitespace.
- Python is meant to be an easily readable language. It is easy to learn.
- Python is slower than other languages but is excellent at interfacing with them to write nice **user code**.

- Python's name is derived from the British comedy group Monty Python, whom Python creator enjoyed while developing the language.

Python

Appeared in	1991; 30 years ago
Designed by	Guido van Rossum
Stable release	3.10.2 ~~(& 2.7.18)~~
URL	http://www.python.org/
OS	cross-platform

Next Steps

Like a real-life language, one needs to practice to gain experience with Python. Luckily there are plenty of resources online to achieve this.

See for example https://www.practicepython.org

The documentation is a great reference: https://docs.python.org/3/

Enjoy programming in Python!

Thank you