DEPARTMENT FOR CONTINUING EDUCATION

UNIVERSITY OF OXFORD

Python for Data science

28-January-2020

Sepi Chakaveh & Assel Altayeva

Overview Data Science

Python

- High level scripting language
- Developed in 1990s by Guido van Rossum
- Emphasis on code readability

Python features

- Interpreted language
  - Python is processed at runtime by the interpreter; there is no need to compile the program before executing it
- Dynamic language
  - No type declarations (of variables, parameters, functions, methods)
    - A name can refer to any type of object
    - Types are bound to values not variables
    - No compile-time type checking of the source code
  - This leads to Python code being short and flexible
- Strongly typed (you cannot add a number to a string)
  - It tracks the types of all values at runtime
  - It flags code that does not make sense as it runs (types, names, etc. are checked when that line runs)
- Supports functional, structured programming methods, Object Oriented Programming
- Supports automatic garbage collection
- Libraries (130,000+ packages)

Python features

- Easy to learn
  - Few keywords, simple structure, and a clearly defined syntax
- Easy to read
  - The language definition enforces code structure that is easy to read
  - Python code is more clearly defined
  - Syntax leverages indentation

In C

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
}
```

In Java

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

In Python

```python
print('Hello World!')
```

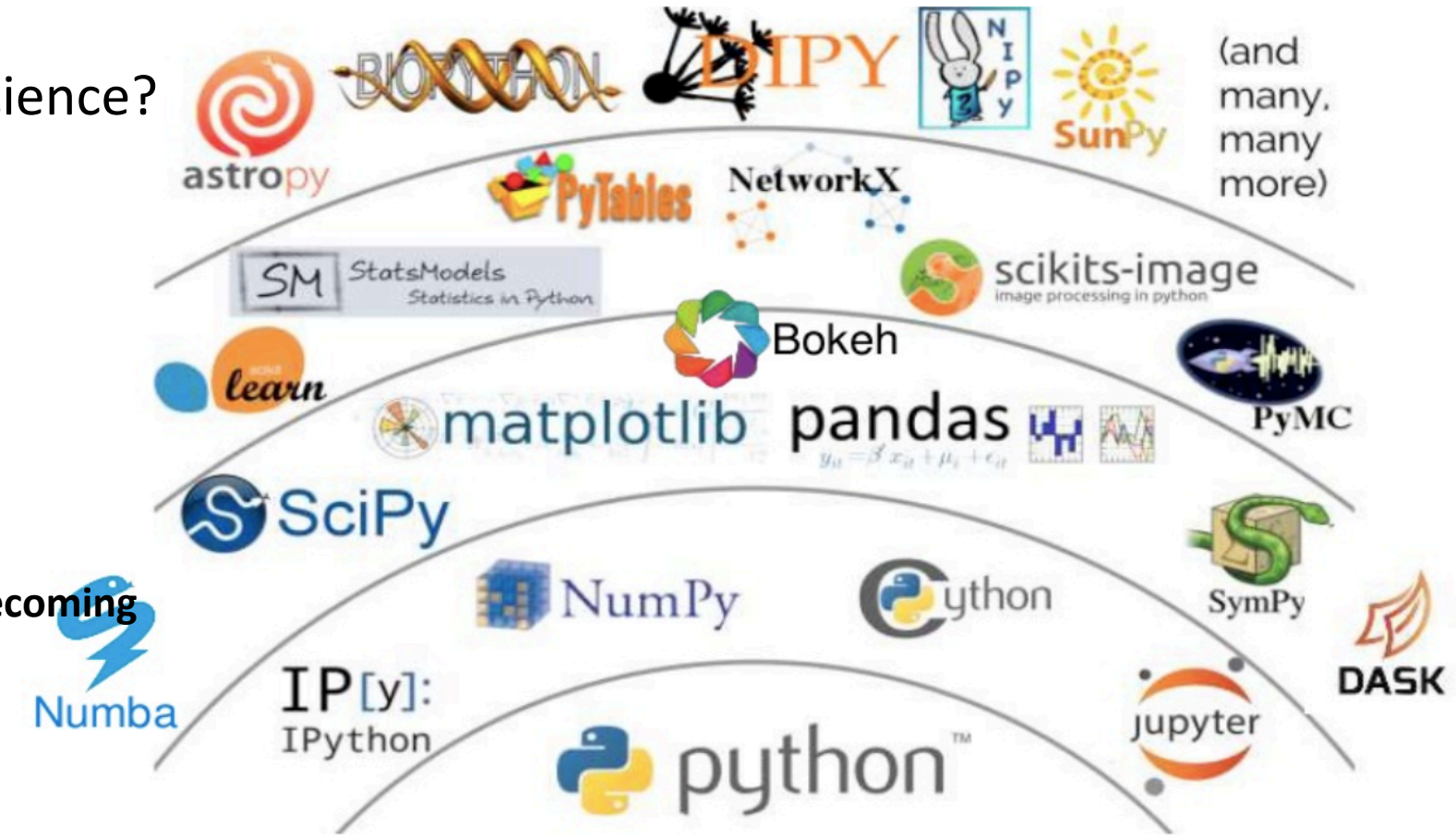
Python ecosystem

Why Python for Data Science?
Why not R, Julia, Go?

- **Design goal**
- **Packages**
- **Speed**

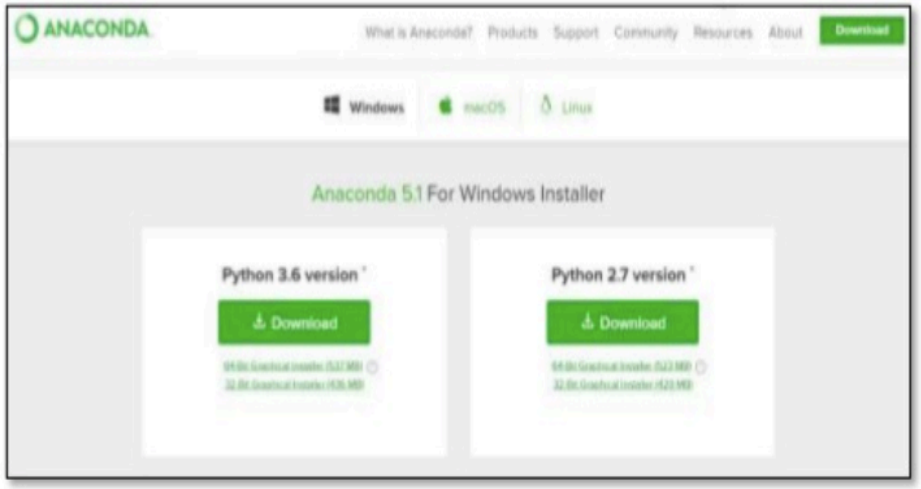
Python and its packages are becoming popular choice

- Quality
- Transparency
- Reproducibly
- Collaboration



Installation

Anaconda comes with pre-installed libraries
In this tutorial, we will be working on Jupyter notebook using Python 3



- Go to: http://continuum io/downloads
- Scroll down to download the graphical installer suitable for your operating system

After successful installation, you can launch Jupyter notebook from Anaconda Navigator

Basic language elements

Python features

- Multi line statements
  - Explicit line continuation using the character \

```
a = 1 + 2 + \
    3 + \
    4 + 5
```

  - Implicit line continuation (implied inside parentheses ( ), brackets [ ] and braces { })

```
b = (1 + 2 +
    3 +
    4 + 5)
c = ['red',
        'yellow',
            'blue']
```

  - Multiple statements in a single line using semicolons

```
a = 1; b = 2; c = 3
```

Python keywords

- Keywords are reserved words
  - are case sensitive
  - used to define syntax and structure
  - cannot be used to name variables, functions, etc.

```
import keyword
print (keyword.kwlist)
```

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

- None represents the absence of a value or a null value
- None has data type NoneType

Variables

- A variable (var) is a name that refers to a value
- The name points to a memory address where the value is stored
- Variables allow us to store and reuse values
- We *assign* a value to a var using the operator =
  - e.g. `number_hours_per_day = 24`
  - It evaluates the expression on the right hand side (memory address) and stores it on the left hand side
- We can modify values (or variables to which a value was assigned) using methods
  - A method is like a function but it runs on an object
  - `value.method_name()`
  - `variable_name.method_name()`
- Use operator `del` to delete a variable

```
var = 'Hello'
del var
print (var) ## throws a NameError as var is not defined anymore
```

Variables

- Dynamic name binding
  - The identifier is not preceded by a type
  - The identifier is bound to an object that has a type
  - The identifier can be rebound to another object of a different type

```
var = 558

var = 'Python'

var = ['p', 'y', 't', 'h', 'o', 'n']
```

Operators

- Arithmetic operators: `+`, `-`, `*`, `/` (the result is a float), `%` (modulus), `//` (floor division, whole number adjusted to the left), `**` (exponent)
- Assignment operators: `+=`, `-=`, `*=`, `/=`, `%=`, `//=`, `**=`
  - there is no ++ operator

```
x = 5
x += 1
print (x) ## 6
x -= 10
print (x) ## -4
```

- Identity operators: `is`, `is not`
  - check if two values are equal or if vars are located in the same part of the memory

```
a = 'Python' ; b = 'Python'
print (a is not b) ## False as the strings are equal and identical
```

  - two vars that are equal does not imply that they are identical

```
x = [5, 5, 8] ; y = [5, 5, 8]
print (x is y) ## False as they are located in different parts of memory
```

Python mathematical functions

- `round(number[, ndigits])` – rounds the number (can specify precision in the second argument)
- `pow(a, b)` – returns a to the power of b
- `abs(x)` – returns the absolute value of x
- `max(x1, x2, …, xn)` – returns the largest value among variables
- `min(x1, x2, …, xn)` – returns the smallest value among variables

- To use any of the following functions, you need to import the math module

```
import math
```

- `ceil(x)` – rounds the number up and returns its nearest integer
- `floor(x)` – rounds the number down and returns its nearest integer
- `sqrt(x)` – returns the square root of x
- `sin(x)` – returns sin of x where x is in radian
- `cos(x)` – returns cos of x where x is in radian
- `tan(x)` – returns tan of x where x is in radian

Booleans

- Relational operators: `<, >, <=, >=, !=, ==`
  - Equality operator `==`
  - Not equals operator `!=`
  - Comparison operators `<,>,<=,>=` that return `True` or `False`
- Python will stop evaluating a boolean expressions once the answer is clear
  - `False and True and True and True and True`
- Can combine arithmetic, boolean, and relational operators
  - `1+4 < 2+5` (`True`)
  - `9 == 10` (`False`)
- Example

```
a = 100
b = 200
print (a and b) ## 200
```

  - `x and y` returns x if x "false" and y is x is "true"
  - `x or y` returns x if x is "true" and y is x is "false"

String manipulation

- Overloading + and * operators
  - Called overloaded operators because they mean different things for different types (numbers vs strings)
  - Concatenation
    - `'la' + 'land'` results in `'la land'`
  - Multiplication
    - Acts as a copy of the string
    - `'la ' * 2` results in `'la la '`
  - Concatenation + Multiplication combined
    - `'la ' * 2 + 'land'` results in `'la la land'`
- Strings can be compared using relational operators
- Can check if a substring is in a string using `in` (membership operator)
  - `'land'` in `'la la land'` evaluates to `True`
  - Can use `not in` to check that a substring is not in a string

String functions and methods

- `len(s)` – returns the length of the string
- `str(x)` – converts a variable `x` to a string
- `s.lower()` / `s.upper()` – converts all chars in string to lower / upper case
- `s.isalpha()` / `s.isdigit()` / `s.isspace()` – tests if all chars are alphabetic / digits / spaces
- `s.startswith(s2)` / `s.endswith(s2)` – tests if the string starts / ends with `s2`
- `s.strip()` – returns a string with leading and trailing whitespaces removed
  - Can also specify the characters to be removed `s.strip(chars)` but when the combination of `chars` mismatches the char of `s` in the left / right, it stops removing the leading / trailing chars
- `s.find(substring)` – returns the index of the first char matching `substring` or -1 if not found
- `s.replace(old, new)` – returns a new string with all instances of `old` replaced by `new`
- `s.split(delim)` – returns a list of substrings separated by the given delimiter
  - `s.split()` splits on all whitespace chars
  - `s.split(delim, max)` specifies how many splits to do
- `s.join(lst)` – joins the elements in the list `lst` together using the string `s` as the delimiter
  - `'--'.join(['aa', 'bb', 'cc'])` gives `'aa--bb--cc'`

Resources

- Python documentation
  - https://www.python.org/doc/
- Python package index
  - https://pypi.org/
- PEP 8 Style guide
  - https://www.python.org/dev/peps/pep-0008/

The Zen of Python

```
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!
```

Collections

Collections

- List

```
universities = ['ICL', 'UCL', 'KCL', 'Oxford', 'Bath']
```

- Tuple

```
numbers = (1, 2, 3)  ##tuple
```

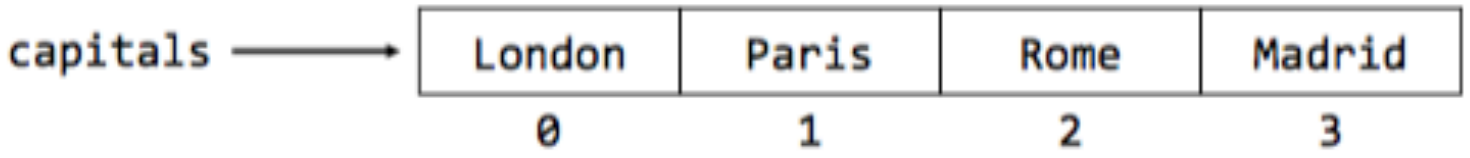
- Set

```
vowels = {'a', 'e', 'i', 'o', 'u'}
```

- Dictionary

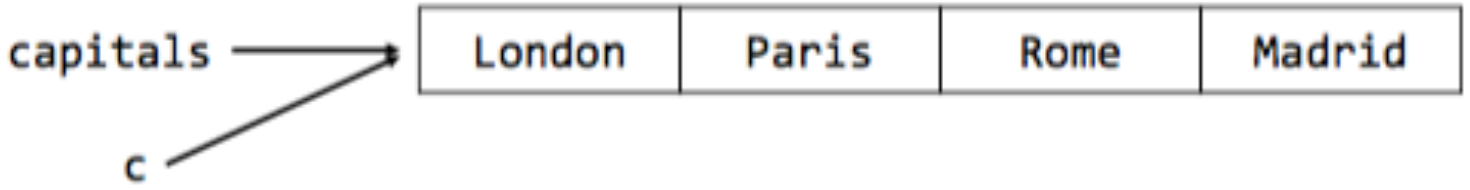
```
alphabet = {'a':'alpha', 'b':'bravo', 'c':'charlie', 'd':'delta'}
```

Lists (teaser)




```
capitals = ['London', 'Paris', 'Rome', 'Madrid']
print (capitals[0]) ## London
print (capitals[2]) ## Rome
print (len(capitals)) ## 4
c = capitals ## Does not copy the list
```

- Assigning the list to a var does not make a copy but makes the two vars point to the list in memory



List methods

- Methods - these methods do not return the modified list but just modify the original list
  - `lst.sort()` – sorts the list (does not return it); add argument `reverse=True` to reverse order
    - `lst.sort()` *## correct*
    - `lst2 = lst.sort()` *## WRONG!! sort() returns None*
  - `lst.reverse()` – reverses the list (does not return it)
  - `lst.insert(index, value)` – inserts `value` at the given `index` shifting elements to the right
  - `lst.append(value)` – adds the value to the end of the list (does not return it)
  - `lst.index(value)` – returns the index of the first instance of `value`
    - Throws `ValueError` if `value` is not present in `lst`
    - Use `in` to check if value is present in `lst` without a `ValueError`
  - `lst.remove(value)` – removes the first instance of `value`
    - Throws `ValueError` if `value` is not present in `lst`
  - `lst.extend(lst2)` – adds the elements of `lst2` to the end of `lst`
    - `extend()` is similar to using list concatenation
  - `lst.count(value)` – counts the number of instances of `value` in the list
  - `lst.pop(index)` – removes and returns the element at the given index
    - `lst.pop()` – removes and returns the last item of the list
  - `lst.clear()` – empties the list

Sorting

- An alternative to `sort()` function is `sorted(list_name)`

- `sorted()` can take as input any iterable
  - `sorted(list_name)` returns a new list with the elements in sorted order

  - The original list is not changed

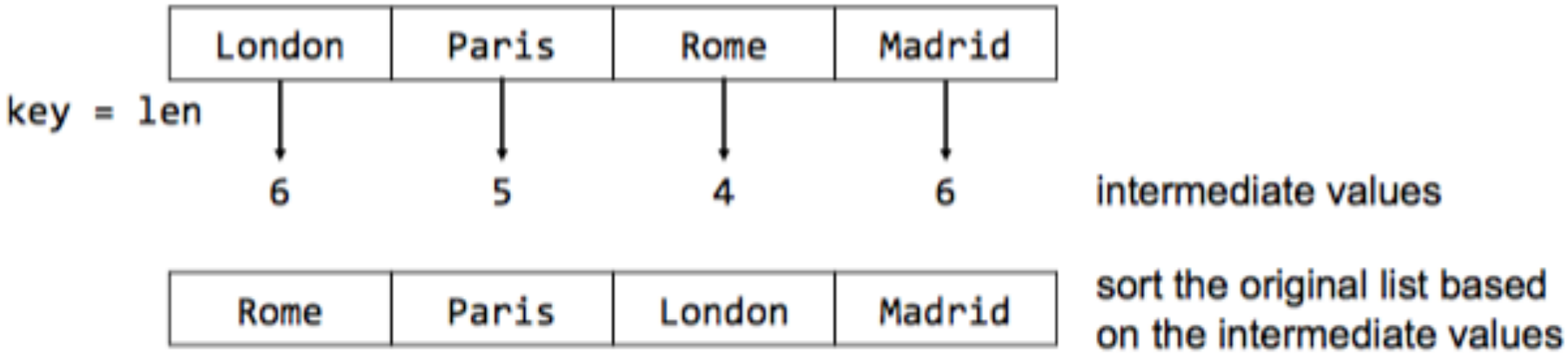
```
lst = [5, 2, 1, 3, 4]
print (sorted(lst))   ## [1, 2, 3, 4, 5]
print (lst)           ## [5, 2, 1, 3, 4]
```

  - Can take the optional argument reverse
    - `print (sorted(lst, reverse=True)) ## [5, 4, 3, 2, 1]`

Custom sorting

- `sorted()` can take the optional argument `key` (applied to tuples and dictionaries too)
- `key` is a function that transforms each element, the new values being used for comparison within the sorting function
- Example

```
capitals = ['London', 'Paris', 'Rome', 'Madrid']
print (sorted(capitals, key=len)) ## ['Rome', 'Paris', 'London', 'Madrid']
```



Sublists

- Use list slicing to refer/assign to a sublist
  - `y=x[i:j]` gives a list `y` with the elements from `i` to `j`-1 inclusive
    - `y` is a new list so that it is not aliased with `x`
  - `x[:i:j]` – equivalent to `x[0:i:j]`, from `start` (0 in this example or a specified index) to `stop` (index `i` exclusive), select every j^{th} element (called `step` parameter)
  - `x[:]` – returns a list that contains all the elements of the original list (it is a copy of the original list, they do not reference the same object)
  - `x[i:]` – returns a list that contains the elements from `i` to the end of the list
  - `x[:j]` – returns a list that contains the elements from the beginning to `j`-1
  - `x[::i]` – returns a list containing every i^{th} element
  - Slicing creates new lists but the items in the lists are aliases to the original elements
    - Might be changing both lists rather than only one
  - Works with negative indices too
    - `x[-i:]` – returns the last `i` elements of the list
    - `x[:-i]` – returns all but the last `i` elements of the list
    - `x[::-1]` – returns the list in reversed order

Control flow

If statement

- How to define an `if` statement in Python

```
if condition:
    block
```

- Example of if statement

```
if temperature > 40:
    print ("it is very hot outside")
```

- `If` statement explained
  - `condition` is a boolean expression followed by `:` (all control flow statements are followed by `:`)
  - `block` represents a sequence of python statements (the block must be indented)
  - `block` is executed only if `condition` evaluates to `True`
- The body of `if` statement is indicated by the indentation with the first unindented line marking the end (applies to the other types of control flow statements)

If-else statement

- How to define an `if-else` statement in Python

```
if condition:
    block if
else:
    block else
```

- Example of `if-else` statement

```
if temperature > 40:
    print ("it is very hot outside")
else:
    print ("it is warm outside")
```

- `If-else` statement explained
  - `block else` is executed only if `condition` evaluates to `False`
- `If-else` statement can have only one `else` block

If-elif-else statement

- Checking multiple expressions with `elif`

```
if condition1:
    block1
elif condition2:
    block2
else:
    block3
```

- Example of `if-elif-else` statement

```
if temperature > 40:
    print ("it is very hot outside")
elif temperature > 20:
    print ("it is warm outside")
elif temperature > 10:
    print ("it is cold outside")
else:
    print ("it is freezing outside")
```

- Conditions are evaluated in order (`else` is optional) and only one block is executed

For loop

- Use a `for` loop if you want to
  - repeat an action a fixed number of times
  - apply an operation to every element in the list (or other iterable)
- How to define a `for` loop in python

```
for item in list_name:
    block
```

- Example of `for` loop

```
for item in letters:
    print (item)
```

- `For` loop explained
  - `item` is the var that takes the value of the item inside the sequence on each iteration
  - `block` is applied to `item` (`letters[0]`)
  - `block` is applied to `item` (`letters[1]`)
  - …
- Do not add or remove from the list during iteration

Looping over lists and dictionaries

- Cannot modify elements in the list using the following definition as `item` is immutable

```
for item in list_name:
    block
```

- To modify elements in the list use the indices of the list

```
for i in range(len(list_name)):
    block
```

- Example

```
for i in range(len(letters)):
    letters[i] = letters[i].lower()
```

- To loop over dictionaries (for dictionaries the `for` loop iterates on keys by default)

```
for key in letters_dict:
    print (d[key])
```

  - or

```
for (key, values) in letters_dict.items():
    print (values)
```

List comprehension

- A compact way to write an expression for a list (works for dictionaries and sets too)

- Syntax for list comprehension

```
[expr for var in list_name]
```

- List comprehension explained
  - `for var in list_name` is similar to a standard for loop (except `:` is missing at the end)
  - `expr` is evaluated for each element `var` giving the values of the new list

- Example

```
capitals = ['London', 'Paris', 'Rome', 'Madrid']
lengths = ([len(c) for c in capitals])
print (lengths)   ## [6, 5, 4, 6]
```

While loop

- Use a `while` loop if you want to repeat an action until a condition is met
- How to define a `while` loop in Python

```
while condition:
    block
```

- Example of `while` loop

```
while x < 10:
    print (x)
    x += 1  ## update counter
```

- `While` loop explained
  - `condition` is a boolean expression
  - if `condition` evaluates to `True` then the `block` is executed otherwise it is skipped
- The value of the counter variable needs to be increased in the body of the loop otherwise it will result in an infinite loop

Functions

Functions

- In math we define a (square) function as follows: `f(x) = x^2`
- In Python we write

```
def f(x):
    return x**2
```

- How to define a function in python

```
def function_name(parameters):
     block
```

- Function definition explained
  - `def` is a python keyword (recap: keywords cannot be used for naming functions or variables)
  - `function_name` is the name that uniquely identifies the function
  - `parameters` are variables (a function can have any number of parameters or none)
  - `block` represents a sequence of python statements (the block must be indented)
  - `return` is a keyword (it means the function will return a value to the caller; if absent then the function returns the special value `None`)
    - `return` returns a value from the function but also immediately terminates the function's execution

Recursion

- A function that calls itself is called a recursive function

- Example
  - Factorial function: x! = 1*2*3*…*x

```
def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
```

base condition

a call to itself

- A recursive function must have a base condition otherwise the function calls itself infinitely

Object Oriented Programming

Object oriented programming (OOP)

- Python is an object-oriented language
  - Various types of objects seen already: `int`, `str`, `list`, `tuple`, `dict`, etc.
- OOP is about defining different types of objects that encapsulate data and that have functions that permit access and manipulation of this data
  - We can define our own types called classes
  - With classes we can create instances (called objects) of that class
  - The difference between a *type* (`str, int`) and a *value* (`"grade", 10`) is analogous to the differences between a *class* and an *object*
- An object has attributes (characteristics) and behavior
- Example
  - An object representing a person has attributes such as name, age, address, and behaviours such as talking, studying
  - An object representing an email has attributes such as subject, body, recipient list, and behaviours such as sending, adding attachments

Classes

- How to define a class

```
class Class_name():
    block
```

- Class definition explained
  - `class` is a keyword which defines an empty class `Class_name`
  - `object` is a type
- By convention, class names start with capital letter
- Classes should have docstrings that describe what the class does
- How to create a new instance of that class (an object which is a copy of the class with actual values)

```
class_instance = Class_name()
```

  - By convention, instances start with a lower case letter
- To delete objects (or attributes) use `del` operator

Class methods

- We can add methods to class objects which are like functions but the first argument to each method is `self`
  - The keyword `self` is always the first parameter in a method
  - Methods will work on each class object
  - When calling the method, `self` is left out (it is not in the parameters)
    - This is because when an object calls its method, the object itself is passed as the first argument
    - `object.function()` translates to `ClassName.function(object)`
    - Calling a method with n arguments is equivalent to calling the function with the method's object as first argument followed by the n arguments
- By convention, methods start with a lower case letter
- Example to change an object's information in the class module

```
def do_something(self, parameter):
    block
```

- Example to call the method on an instance

```
instance.do_something(parameter)
```

Classes example 1

```
class Employee:

    employee_count = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.employee_count += 1

    def display_number_employees(self):
        print ("Total Employees %d" % Employee.employee_count)

    def display_employee_details(self):
        print (f'{self.name} has salary {self.salary}')
```

class attribute (its value is shared among all instances of this class)

instance attributes (specific to each object)

method (a function defined inside the class that defines a behaviour of an object)

Classes example 1

```
emp = Employee("Alice", "90,000")
emp.display_number_employees()      ## Total Employees 1
emp.display_employee_details()      ## Alice has salary 90,000
emp2 = Employee("Bob", "80,000")
emp2.display_number_employees()     ## Total Employees 2
emp2.display_employee_details()     ## Bob has salary 80,000
```

instance of `Employee` class (`emp` is a reference by value to the new object)

- Accessing class attributes

```
print (Employee.employee_count) ## 2
```

dot operator with class name

- Accessing instance attributes

```
print (emp.name) ## Alice
```

dot operator with object

Object oriented programming

- Principles of OOP:
  - Inheritance
    - Use details from a new class without modifying an existing class
  - Encapsulation
    - Hide the private details of a class from other objects
  - Polymorphism
    - Use common operations in different ways for different data inputs

NumPy

NumPy

- Package for scientific computing
  - Has support for (powerful) N-dimension array objects that describe collections of items of the same type
  - The array class is called `ndarray` (the alias is `array`)
    - Not to be confused with the Python array

- Difference between lists and ndarrays
  - The elements in a NumPy array must have the same type
  - NumPy is array oriented computing
  - NumPy implements multi-dimensional arrays efficiently
  - NumPy arrays have smaller memory consumption and better runtime compared to Python lists

NumPy

- To import numpy
  - `import numpy as np`
  - We will assume this import and use `np` in all the examples that will follow

- How to create a NumPy array

```
arr = np.array([1, 2, 3])
print (arr)       # [1, 2, 3]
print (type(arr)) # <class 'numpy.ndarray'>
```

  - In NumPy dimensions are called axis
    - `arr` has one axis with 3 elements

- NumPy axes
  - Axis 0 runs vertically downwards across rows
  - Axis 1 runs horizontally across columns

NumPy arrays

- NumPy has several functions to create arrays with placeholders when the size is known
  - This minimizes the need of growing arrays which is an expensive operation
  - Unless specified, the default `dtype` is `float64`
  - `np.zeros((i,j))  # array (of i rows and j columns) full of zeros`
  - `np.ones((i,j,k)) # array (of i arrays of j rows and k columns) full of ones`
  - `np.empty((i,j), dtype=int) # array with random content (output may vary)`
- NumPy has a function similarly to `range` to create sequences of numbers
  - When using ints
    - `np.arange(start, stop, step)` where `start` is optional (default is 0) and `step` is optional (default is 1)
  - When using non-ints the results may not be consistent due to the finite floating point precision
    - `np.linspace(start, stop, num=50, endpoint=True)` where `num` is the number of samples to generate and `endpoint` is a boolean that determines if `stop` is the last sample
- Use `np.zeros_like(arr)`, `np.ones_like(arr)`, `np.empty_like(arr)` to create an array with the same shape and type as a given array (i.e. `arr`)

Pandas

Download Jupyter notebooks for the Pandas exercise

https://github.com/asel-datascience/ODS2020/tree/master/Week_1_Intro_Python

Pandas

- Pandas is the most important package for data science and data analysis
  - It offers powerful and flexible data structures that make data manipulation and analysis easy
  - Pandas is used to clean, transform, and analyse data
  - The name comes from the term panel data which describes multidimensional data involving measurements over time
  - Pandas is built on top of NumPy package
- To import pandas

```
import pandas as pd
```

  - We will assume this import and use `pd` in all the examples that follow

Pandas main components

- `DataFrame` – represents a multidimensional table consisting of a collection of `Series`
- `Series` – represents a column
- A data frame can be seen as a way of representing data in rectangular grids where
  - Each row represents values of an instance
    - The difference between rows is indicated by the `Index`
  - Each column is a vector containing data for a specific variable
    - The difference between columns is given by the column names

`Series`

	country
0	UK
1	France
2	Italy

+

`Series`

	capital
0	London
1	Paris
2	Rome

=

`DataFrame`

	country	capital
0	UK	London
1	France	Paris
2	Italy	Rome

Index:
`0`
`1`
`2`

Column names:
`country`
`capital`

Creating Series

- You can create a `Series` with the constructor `Series(data, index, dtype)`
  - `data` can be a ndarray, list, dictionary
  - `index` values must be unique and must have the same length as data (default `np.arange(n)`)
  - `dtype` represents the data type (if not specified, then it is inferred from data)

- Example

```
s1 = pd.Series(np.array(['a','b','c','d']))
print (s1)

s2 = pd.Series(np.array(['a','b','c','d']),
                    index=[10, 11, 12, 13])
print (s2)
```

```
0    a
1    b
2    c
3    d
dtype: object

10    a
11    b
12    c
13    d
dtype: object
```

Creating Series

- Creating `Series` from a dictionary
  - If no index is specified, the dictionary keys are used to construct the `Index`

```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s1 = pd.Series(data)
print (s1)
```

```
a    0.0
b    1.0
c    2.0
dtype: float64
```

  - If index is passed, the values in data corresponding to the labels in the index will be used

```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s2 = pd.Series(data, index=['b','c','d','a'])
print (s2)
```

```
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Accessing Series elements

- Accessing elements using the index

```
s = pd.Series(['a', 'b', 'c', 'd', 'e'])
print (s)
print (s[0])      ## a
print (s[:3])
```

```
0    a
1    b
2    c
3    d
4    e
dtype: object
```

```
0    a
1    b
2    c
dtype: object
```

- Accessing elements using the label index (an exception is raised if a label is not found)

```
s = pd.Series([1,2,3,4,5], index = ['a','b','c','d','e'])
print (s)
print (s['a'])    ## 1
print (s[['a','c','d']])
```

```
a    1
b    2
c    3
d    4
e    5
dtype: int64
```

```
a    1
c    3
d    4
dtype: int64
```

Creating DataFrames

- You can create a `DataFrame` from a list

- Example

```
import pandas as pd


arr = [['UK','London'],['France','Paris'],['Italy','Rome']]

df = pd.DataFrame(arr, columns=['country','capital'])

print (df)
```

← DataFrame constructor

```
   country    capital
0       UK     London
1   France      Paris
2    Italy       Rome
```

Index of the DataFrame →

Creating DataFrames

- You can create a `DataFrame` from a `NumPy` array
  - Need to specify `data` in the constructor
- Example

```
import numpy as np
import pandas as pd

arr = np.array([['', 'country', 'capital'],
                [0, 'UK', 'London'],
                [1, 'France', 'Paris'],
                [2, 'Italy', 'Rome']])

df = pd.DataFrame(data=arr[1:, 1:], columns=arr[0,1:])
print (df)
```

```
   country  capital
0       UK   London
1   France    Paris
2    Italy     Rome
```

Creating DataFrames

- You can create a `DataFrame` from a dictionary
  - The data in the dictionary needs to be organised for `pandas`
  - Each key, value item in the dictionary will correspond to a column in the `DataFrame`

- Example

```
import pandas as pd

data_dict = {'country': ["UK", "France", "Italy"],
             'capital': ["London", "Paris", "Rome"]}

data = pd.DataFrame(data_dict)
print (data)
```

```
   country  capital
0       UK   London
1   France    Paris
2    Italy     Rome
```

Creating DataFrames

```
data_dict = {'country': ["UK", "France", "Italy"],
             'capital': ["London", "Paris", "Rome"]}
data = pd.DataFrame(data_dict)
print (data)
```

```
   country  capital
0       UK   London
1   France    Paris
2    Italy     Rome
```

- The Index of the `DataFrame` defaults to indices

- You can define custom values for the `Index`

```
data = pd.DataFrame(data_dict, index=['+44','+33','+39'])
print (data)
```

```
     country  capital
+44       UK   London
+33   France    Paris
+39    Italy     Rome
```

  - Alternatively you can do `data.index = ['+44','+33','+39']`

DataFrame operations

- We will be looking at the IMDB movie dataset (https://www.kaggle.com/PromptCloudHQ/imdb-data)
- To print the first few rows use the function `head()`
  - `head()` outputs the first 5 rows of the DataFrame by default
  - You can also set the number of rows you want to output by passing an integer as a parameter
    - For example `head(10)` outputs the first 10 rows

```
df = pd.read_csv("IMDB-Movie-Data.csv", index_col="Rank")
print (df.head())
```

The Rank column from the data is set as the `Index`

Rank	Title	Genre	Description	Director	Actors	Year	Runtime (Minutes)	Rating	Votes	Revenue (Millions)	Metascore
1	Guardians of the Galaxy	Action,Adventure,Sci-Fi	A group of intergalactic criminals are forced ...	James Gunn	Chris Pratt, Vin Diesel, Bradley Cooper, Zoe S...	2014	121	8.1	757074	333.13	76.0
2	Prometheus	Adventure,Mystery,Sci-Fi	Following clues to the origin of mankind, a te...	Ridley Scott	Noomi Rapace, Logan Marshall-Green, Michael Fa...	2012	124	7.0	485820	126.46	65.0
3	Split	Horror,Thriller	Three girls are kidnapped by a man with a diag...	M. Night Shyamalan	James McAvoy, Anya Taylor-Joy, Haley Lu Richar...	2016	117	7.3	157606	138.12	62.0
4	Sing	Animation,Comedy,Family	In a city of humanoid animals, a hustling thea...	Christophe Lourdelet	Matthew McConaughey,Reese Witherspoon, Seth Ma...	2016	108	7.2	60545	270.32	59.0
5	Suicide Squad	Action,Adventure,Fantasy	A secret government agency recruits some of th...	David Ayer	Will Smith, Jared Leto, Margot Robbie, Viola D...	2016	123	6.2	393727	325.02	40.0

DataFrame operations

```
df = pd.read_csv("IMDB-Movie-Data.csv", index_col="Title")
print (df.head())
```

The Title column from the data is set as the `Index`

Title	Rank	Genre	Description	Director	Actors	Year	Runtime (Minutes)	Rating	Votes	Revenue (Millions)	Metascore
Guardians of the Galaxy	1	Action,Adventure,Sci-Fi	A group of intergalactic criminals are forced ...	James Gunn	Chris Pratt, Vin Diesel, Bradley Cooper, Zoe S...	2014	121	8.1	757074	333.13	76.0
Prometheus	2	Adventure,Mystery,Sci-Fi	Following clues to the origin of mankind, a te...	Ridley Scott	Noomi Rapace, Logan Marshall-Green, Michael Fa...	2012	124	7.0	485820	126.46	65.0
Split	3	Horror,Thriller	Three girls are kidnapped by a man with a diag...	M. Night Shyamalan	James McAvoy, Anya Taylor-Joy, Haley Lu Richar...	2016	117	7.3	157606	138.12	62.0
Sing	4	Animation,Comedy,Family	In a city of humanoid animals, a hustling thea...	Christophe Lourdelet	Matthew McConaughey,Reese Witherspoon, Seth Ma...	2016	108	7.2	60545	270.32	59.0
Suicide Squad	5	Action,Adventure,Fantasy	A secret government agency recruits some of th...	David Ayer	Will Smith, Jared Leto, Margot Robbie, Viola D...	2016	123	6.2	393727	325.02	40.0

DataFrame operations

- To print the last few rows use the function `tail()`
  - `tail()` outputs the last 5 rows of the DataFrame by default
  - You can also set the number of rows you want to output by passing an integer as a parameter

```
print (df.tail(3))
```

Title	Rank	Genre	Description	Director	Actors	Year	Runtime (Minutes)	Rating	Votes	Revenue (Millions)	Metascore
Step Up 2: The Streets	998	Drama,Music,Romance	Romantic sparks occur between two dance studen...	Jon M. Chu	Robert Hoffman, Briana Evigan, Cassie Ventura,...	2008	98	6.2	70699	58.01	50.0
Search Party	999	Adventure,Comedy	A pair of friends embark on a mission to reuni...	Scot Armstrong	Adam Pally, T.J. Miller, Thomas Middleditch,Sh...	2014	93	5.6	4881	NaN	22.0
Nine Lives	1000	Comedy,Family,Fantasy	A stuffy businessman finds himself trapped ins...	Barry Sonnenfeld	Kevin Spacey, Jennifer Garner, Robbie Amell,Ch...	2016	87	5.3	12435	19.64	11.0

DataFrame missing values

- Your data may have missing or null values (which are placeholders for missing values)
- To deal with null values you can
  - Remove the rows/columns with nulls
  - Replace nulls with non-null values (this process is called imputation)
- `df.isnull()` – returns an array of bools indicating whether each corresponding element is missing
- `df.isnull().sum()` – returns the number of nulls in each column
- In the movie example

  `df.isnull().sum()`
  - `Revenue_mils` has 128 missing values
  - `Metascore` has 64 missing values

```
Rank            0
Genre           0
Description     0
Director        0
Actors          0
Year            0
Runtime_mins    0
Rating          0
Votes           0
Revenue_mils  128
Metascore      64
```

DataFrame removing missing values

- `df.dropna(axis=0, inplace=False)` – removes null values and returns a new `DataFrame` without modifying the original one
  - Deletes every row with at least a single null value
  - To drop columns with null values set the `axis` argument to 1
  - You can modify in place by setting the `inplace` argument to `True`
- Movie example

```
print (df.shape) ## (1000, 11) is the original shape


temp_df = df.dropna()

print (temp_df.shape) ## (838, 11) as it dropped the rows where Revenue_mils is
null and the rows where Metascore is null


temp_df = df.dropna(axis=1)

print (temp_df.shape) ## (1000, 9) as it dropped the Revenue_mils and Metascore
columns
```

DataFrame operations

- `append(other)` – appends the rows of `other` returning a new object
  - `other` can be a `DataFrame` or `Series`/`dict`-like object, or list of these

- `drop_duplicates(subset=None, keep='first', inplace=False)`

  - `subset` represents the columns used for identifying duplicates
    - Can take the value of a column label or sequence of labels (default uses all columns)

  - `keep` can take one of the following values
    - `'first'` – dropping duplicates except for the first occurrence (i.e. the first row)
    - `'last'` – dropping duplicates except for the last occurrence
    - `False` – dropping all duplicates

  - `inplace` is a boolean that determines whether to drop duplicates in place or to return a copy

File manipulation

Files

- A file is a named location on disk and is used to store data on the hard disk
  - The hard disk is a non-volatile memory in contrast to RAM which is volatile and loses its data when the computer is turned off
- Order of file operations
  - Opening the file
  - Performing operations (read or write)
  - Closing the file
- Python handles two types of files: binary and text files
  - Most files are binary files
    - Image files: .jpg, .png, .bmp, .gif, etc.
    - Documents: .doc, .xls, .pdf, etc.
  - Text files have an unseen character at the end of each line that lets the text editor know that there should be a new line
    - In Python this is denoted by the `'\n'`

Opening a file

- Use `open(file_name, mode)` which returns a file object (denoted `f` in the examples)
  - The file object is also called handle as it is used to read and modify the file
  - The default is to open the file in text mode as we want to get strings when reading the file
  - You can also open the file in binary mode which returns bytes which is useful for non-text files
  - The default encoding is platform dependent (`'utf-8'` for Linux) thus it is recommended that you specify the encoding type for files in text mode

- Example
  - `f = open("test.txt", encoding = 'utf-8')`

Opening a file

- You can specify the mode while opening the file
  - `open(file_name, 't')` – opens in text mode (this is the default)
  - `open(file_name, 'r')` – for reading (this is the default)
  - `open(file_name, 'w')` – for writing (erases the contents of the file)
  - `open(file_name, 'a')` – for appending (keeps the contents of the file; creates a new file if `file_name` does not exist)
  - `open(file_name, '+')` – for updating (reading and writing)
    - `'r+'` is used for making changes to the file and reading information from it
  - `open(file_name, 'b')` – opens in binary mode
    - `'wb'` represents write mode for a binary file
  - `open(file_name, 'x')` – used to create a file
    - If a file `file_name` already exists, the function call will fail

Closing a file

- Not necessary as the garbage collector will clean up unreferenced objects but it is a good idea to free up system resources
- Use `f.close()` to close a file
  - However if an exception occurs when you are operating with the file, the code exits without closing the file
  - The best practice to work with files in Python is to use `with` statement (also known as context manager) which ensures that the file is closed when the block inside `with` is exited (the call to the `close()` method is done internally)
- Example

```
with open("test.txt", encoding = 'utf-8') as f:
    # perform file operations
```

- You can't access a file after closing it
  - Attempting to read from or write to a closed file object throws an exception
  - You need to open the file again if you want to read from or write to it

Reading from a file

- Need to open a file in reading mode (`r`)
- `f.read()` - returns a string that represents the contents of the entire file
  - Not recommended for large files
  - `read()` returns newline as `'\n'`
  - When the end of the file is reached, `read()` returns an empty string on further reading
- To read a fixed number of characters
  - `f.read(x)` – will read `x` characters
  - Calling the function again will start reading from the unread characters
- Example

```
with open("test.txt", 'r', encoding = 'utf-8') as f:
    f.read(4) # read the first 4 chars
    f.read(4) # read the next 4 char
    f.read()  # read the rest of the file
    f.read()  # further reading returns an empty string
```

Writing to a file

- Need to open a file in append (`a`) or write (`w`) mode to write to it
  - Remember that `w` mode will overwrite the contents of the file if the file already exists
- Multiple writes are concatenated
- If you want to write on new lines you need to include the newline character `"\n"`
- If you want to write something that is not a string, you need to cast them to strings
- Example

```
with open("test.txt", 'w', encoding = 'utf-8') as f:
    f.write("First line\n\n")
    f.write("Second line\n")
```

Challenge for the next week

- Go through tutorial that loads data from a text file and manipulates it

https://github.com/asel-datascience/ODS2020/blob/master/Week_1_Intro_Python/Olympic_loops.ipynb

- For any questions, suggestions and discussions email or join the course slack channel:
- https://join.slack.com/t/overviewofthe-gsm9095/shared_invite/enQtOTE3MDYzNDYyMTY3LWVkOGUxZGFhMWU5N2VhN2VhZDhkNjgzN2UwOGY1NGExMWM0MWViZWY4ZWE2YWQxZTE4Mzg2MTVjM2ZkZTBmNWY

- Tutorial jupyter notebooks uploaded here:

https://github.com/asel-datascience/ODS2020