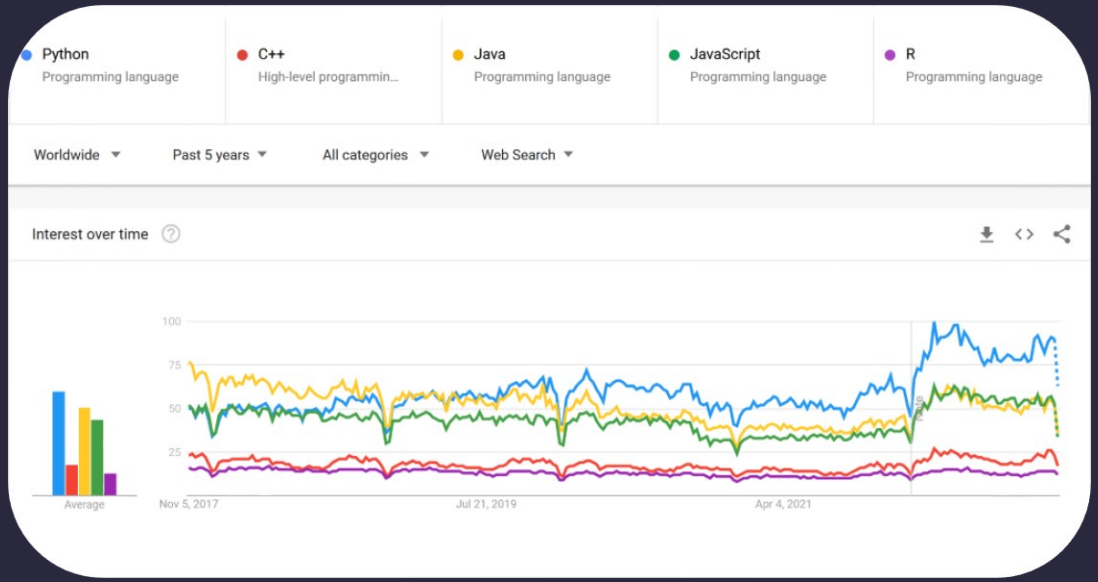
Crash Course In Python

Waleed Esmail
GSI Helmholtzzentrum für Schwerionenforschung

Python Popularity



* Google trends

JÜLICH
FORSCHUNGSZENTRUM

Python vs C++

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

```cpp
#include <iostream>
using namespace std;

int main(){
  cout << "Hello world! \n";
  return 0;
}
```

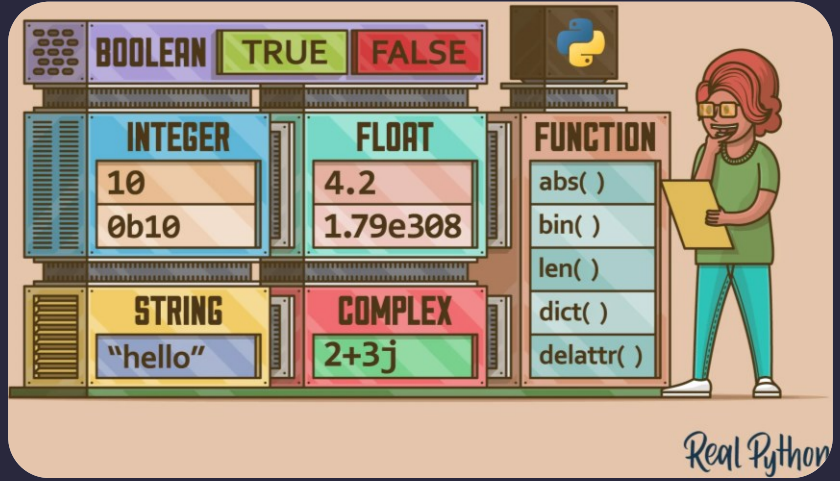
The Zen Of Python

```
In [1]: 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!
```

Table Of Contents

/01

Data Types



Data Types I

In **Python** DATA are objects. An object is a chunk of data that contains:

1. *A type*
2. *A unique ID*
3. *A value*
4. *A reference count*

Type	Example	Is Mutable ?
bool	True, False	no
int	1, 1000, 545477	no
float	3.14, 5.4e6	no
complex	3j, 5 + 9j	no
str	“hello”, ‘world’	no
list	[1,2,88]	yes
tuple	(4.2, 9)	no
dict	{“myKey”: 6}	yes

Data Types II

In Python if you want to know the type of anything, you can use the built-in method type():

```
In  [1]: x = 1

In  [2]: type(x)
Out [2]: <class 'int'>

In  [3]: x = "hello world"

In  [4]: type(x)
Out [4]: <class 'str'>
```

Alternatively, you can use isinstance(type):

```
In  [5]: isinstance(x, str)
Out [5]: True
```

Data Types III

Immutable Objects

```
In  [1]: x = 6
In  [2]: y = x
In  [3]: y = 11
In  [4]: x
Out [4]: 6
```

Mutable Objects

```
In  [1]: x = [6,4,22]
In  [1]: y = x
In  [4]: y[0] = -1
In  [4]: x
Out [4]: [-1,4,22]
```

Data Types IV

A Python string is a sequence of characters are objects

```
In  [1]: x = ‘hello world’
In  [2]: y = “hello world”
```

- You can also use three single quotes (‘’’) or three double quotes (“””)
- Python string indexing works similar to other languages [start:end:step]

```
In  [3]: x[0]
Out [3]: ‘h’

In  [4]: x[0:4]
Out [4]: ‘hell’
```

- You can format string with % or f-strings

```
In  [3]: print(“success percentage %.3f”%98.134343)
Out [3]: success percentage 98.134

In  [3]: print(“success percentage {} % and failure {} %”.format(98.134343, 100-98.134343))
Out [3]: success percentage 98.134343 % and failure 1.865657 %
```

Comments

In **Python**, comments begin with the # character

```
In  [1]: # this is a comment
In  [2]: y = "hello world"
```

- You can also use three single quotes (‘’’) or three double quotes (“””)

```
In [10]: """
    ...: this is a big comment
    ...: """
```

JÜLICH
FORSCHUNGSZENTRUM

Exercise

Rydberg's constant

Rydberg's constant R_∞ for a heavy atom is used in physics to calculate the wavelength to spectral line

The constant has been found to have the following value:

$$R_\infty = \frac{m_e e^4}{8\varepsilon_0^2 h^3 c}$$

where

- $m_e = 9.109 \times 10^{-31}\,\mathrm{m}$ is the mass of an electron
- $e = 1.602 \times 10^{-19}\,\mathrm{C}$ is the charge of a proton (also called the *elementary charge*)
- $\varepsilon_0 = 8.854 \times 10^{-12}\,\mathrm{C\,V^{-1}\,m^{-1}}$ is the electrical constant
- $h = 6.626 \times 10^{-34}\,\mathrm{J\,s}$ is Planck's constant
- $c = 3 \times 10^{8}\,\mathrm{m/s}$ is the speed of light

$$R_\infty = 10961656.2162 \quad (\text{in } \mathrm{m}^{-1})$$

/02 Choose with if

Control Statement if I

In **Python**, **indentation** is used to mark off the sections of code, it *define a program's structure*

```
In [1]: x = input('Please enter a number ')
Please enter a number
```

Taking input from the user
e.g. 10

```
In  [2]: x = int(x)
```

Converting to integer

```
In  [3]: if x>0:
    ...:     print ("The value of {} is greater than 0".format(x))
Out [3]: The value of 10 is greater than 0
```

4 space indentation

```
In [4]: x = int(input('Please enter a number '))
```

Be Pythonista: one-liner
e.g. -1

```
In [5]: if x>0:
   ...:     print ("The value of {} is greater than 0".format(x))
```

Control Statement if II

```
In  [6]: x = int(input('Please enter a number '))
In  [7]: if x>0:
    ...:     print ("The value of {} is greater than 0".format(x))
         else:
             print ("The value of {} is smaller than 0".format(x))

Out [7]: The value of -1 is smaller than 0


In  [8]: x = int(input('Please enter a number '))
In  [9]: if x>0:
    ...:     print ("The value of {} is greater than 0".format(x))
         else:
             print ("The value of {} is smaller than 0".format(x))

In [10]: if x>0:
    ...:     print ("The value of {} is greater than 0".format(x))
    ...: elif x<0:
    ...:     print ("The value of {} is smaller than 0".format(x))
    ...: else:
    ...:     print ("The value of {} is 0".format(x)
```

This time let's enter **0**

Some Python Operators

- You can do multiple comparisons with (or) and (and) operators

```
In  [1]: x, y, z = True, True, False
In  [2]: (x or y) and z
Out [2]: ??

In  [3]: (x and y) or z
Out [3]: ??
```

Simultaneous assignment

- Python membership operator (in)

```
In  [1]: l = [1,3,66,89,0]
In  [2]: 0 in l
Out [2]: True
```

- Python identity operators (is) and (is not)

```
In  [1]: x, y = 10, 20
In  [2]: x is y
Out [2]: False
```

/03
Loops

Loops

Python gives us two choices for reptation (while) and (for)

1. while loop: countdown example

```
In  [1]: counter = 5
In  [2]: while counter >= 0:
    ...:     print (counter, end='')
    ...:     counter -= 1
Out [2]: 5 4 3 2 1 0
```

2. for loop

```
In  [1]: for i in 5,4,3,2,1,0:
    ...:     print (i, end='')
Out [1]: 5 4 3 2 1 0
```

```
In  [1]: for i in range(5, 0, -1):
    ...:     print (i, end='')
Out [1]: 5 4 3 2 1
```

range(start, end, step)

You can always skip with (continue) and cancel with (break)

Use `for` together with tqdm تقدم

- tqdm is a **Python** external library to create simple progress bars
- Usage: `tqdm(iterable)`

```
In  [1]: from tqdm import tqdm
In  [2]: for x in tqdm(range(10000000)):
    ...:     pass
```

```
from tqdm import tqdm

for x in tqdm(range(100000000)):
    pass
```

```
 6%|▋▋▋                                   | 6063227/100000000 [00:02<00:32, 2863933.09it/s]
```

JÜLICH FORSCHUNGSZENTRUM

Exercise

Trapped quantum particle

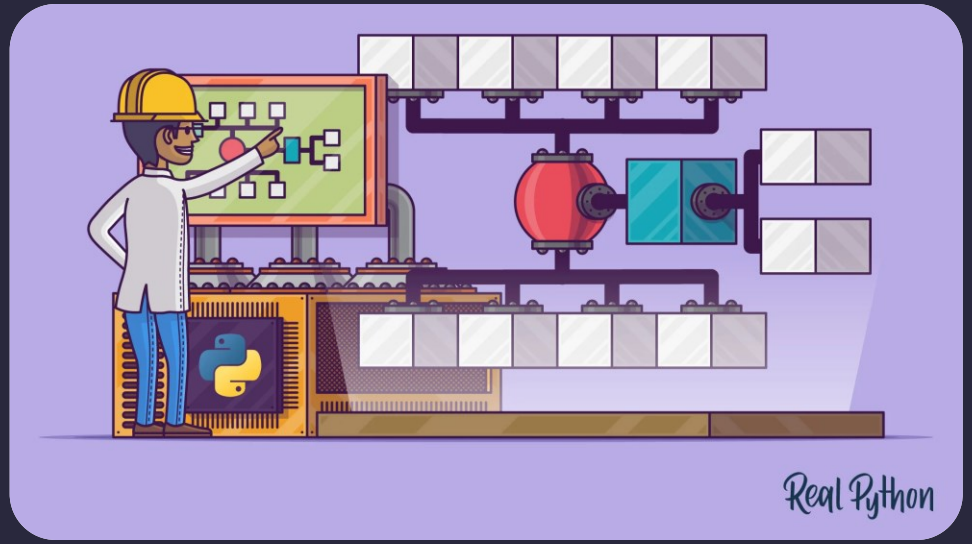
Consider an electron with mass 9.11×10^{-31} kg, trapped in a box of size 10^{-11} m. It starts at the lowest energy-level, E_1 (not E_0!), and jumps upwards, one step at a time, ending up at a much higher energy level, E_{30}. Each step from a level E_i to a level E_{i+1} will have required an energy

$$E_{i+1} - E_i = \frac{((i+1)^2 - i^2)h^2}{8mL^2}$$

Write a for loop which calculates the energy required for each step along the way, and saves them in a list.

where m is the particle's mass and h is Planck's constant, $h = 6.626 \times 10^{-34}$ J s.

/04 Data Structures



Data Structures: Lists

A data structure is a way of organizing data so it can be accessed efficiently
In **Python** there are: lists, tuples, dictionaries and sets

- lists are **used to store multiple items in a single variable**
- lists are **mutable**

```
In  [1]: empty_list = []
In  [2]: another_empty_list = list()
In  [3]: weekdays = [‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’, ‘Friday’]
In  [3]: randomness = [‘hello’, 2, 11.2e8, dict() ]
```

Data Structures: List Operations

```
In  [1]: a = list(range(10))

1. Slicing/indexing: a[start:end:step]
In  [1]: a[0:6:2]
In  [2]: a[-1]

2. Add items: append(item), insert(loc, item), extend(list)
In  [1]: a.append(10)
In  [2]: a.insert(2, -1)
In  [3]: a.extend(list(range(10, 20)))


3. Delete items: del, remove(item)
In  [1]: del a[0]
In  [2]: a.remove(1)



4. Reorder items (sorts the list itself, in place): sort(list)
In  [1]: a.sort()
```

Iterate over many lists simultaneously

```
In  [1]: days = ['Monday', 'Tuesday', 'Wednesday']
In  [2]: fruits = ['banana', 'orange', 'peach']
In  [3]: drinks = ['coffee', 'tea', 'juice']
In  [4]: for day, fruit, drink in zip(days, fruits, drinks):
    ...:     print(day, ": drink", drink, "- eat", fruit)
```

Data Structures: Dictionaries

- **Dictionaries** consists of key and value, also called associative arrays or hash maps
- The order of items doesn't matter
- Items are selected by **unique keys**
- **Keys** must be **immutable.**

```
In  [1]: empty_dictionary = {}
In  [2]: empty_dictionary = dict()
```

- Usage: `dict_name = {"key": value}`

```
In  [3]: python_creator = {"firts": "Guido", "middle": "van", "last": "Rossum" }
In  [4]: python_creator = dict(firts: "Guido", middle: "van", last: "Rossum" )
```

How to get an item: `dict_name[key]`

```
In  [5]: python_creator["last"]
Out [5]: Rossum
```

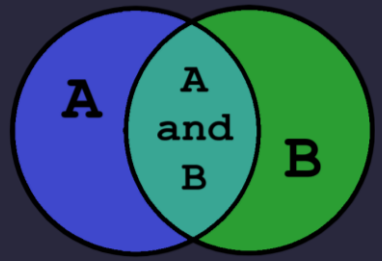
How to Iterate over Dictionaries

```
In  [1]: for key, value in python_creator.items():
    ...:     print(key, value)

In  [2]: for key in python_creator.keys():
    ...:     print(key)

In  [3]: for value in python_creator.values():
    ...:     print(value)
```

Data Structures: Sets

A set is a collection of **unique unordered items**

```
In  [1]: empty_set = set()
```



- You can convert a list to a set:

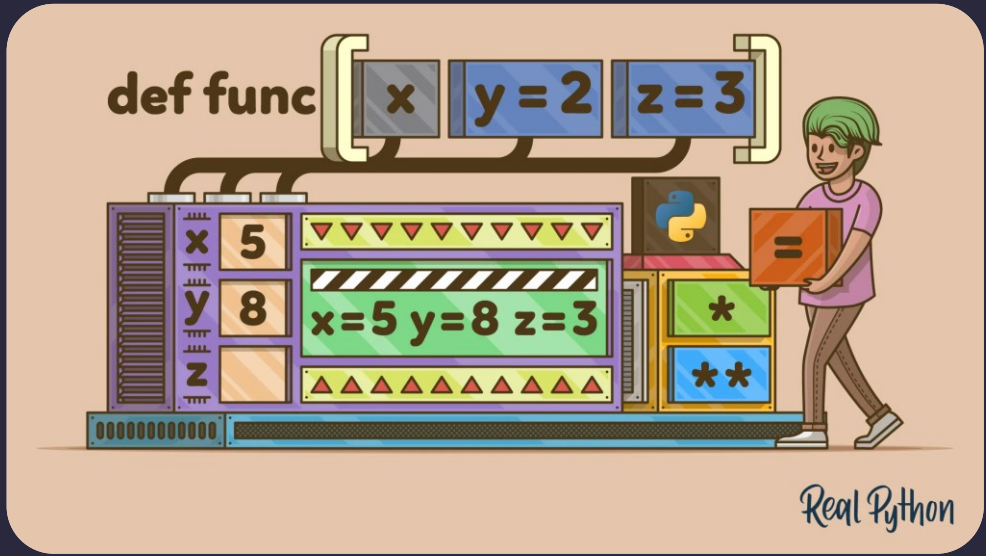
```
In  [2]: a = [1,1,4,56,9,0,9]
In  [3]: set(a)
Out [3]: {0, 1, 4, 9, 56}
```

- Operations on sets: Intersection (&), Union (|), difference (-), subset (<=), ...

An example from track reconstruction:

```
In  [1]: reco_track = set(hit_ids)
In  [2]: true_track = set(hit_ids)
In  [3]: true_track & reco_track
Out [3]: what hits_ids in both reco and true
```

/05 Functions



Functions I:

- A function is a named piece of code. It can take any input parameters and return any number of outputs
- **Define and call**

```
In  [1]: def do_something():
    ...:     pass
```

- **Positional arguments**

```
In  [1]: def simple_calculator(x, y):
    ...:     return {“sum” :x+y, “difference”:x-y, “multiplication”:x*y, ”division“:x/y}
In  [2]: simple_calculator(1.1, 23.0)
```

- **Keyword arguments**

```
In  [3]: simple_calculator(x=1.1, y=23.0)
In  [4]: simple_calculator(y=1.1, x=23.0)
```

Explode positional/keyword arguments

- **Python** doesn't have pointers

```
In  [1]: def print_args(*args):
    ...:     print("Arguments: ", args)
In  [2]: print_args(1,"text", 1.4e8, dict(), tuple(), set())
Out [2]: Arguments:  (1, 'text', 140000000.0, {}, (), set())
```

- **Python** print() function is an obvious application
- In addition, you can use two asterisks (**) to group keyword arguments into a dictionary

```
In  [1]: def print_kwargs(**kwargs):
    ...:     print("Arguments: ", kwargs)

In  [2]: print_kwargs(1,"text", 1.4e8, dict(), tuple(), set())

In  [3]: print_kwargs(a=1, b="text", c=1.4e8, d=dict(), e=tuple(), f=set())
Out [2]: {'a': 1, 'b': 'text', 'c': 140000000.0, 'd': {}, 'e': (), 'f': set()}
```

Functions II: Docstrings

You can write documentation for any **Python** function or **class**

```
In  [1]: def simple_calculator(x, y):
             '''
             This function implements a very simple calculator
             Parameters:
                  x (float): the first number
                  y (float): the second number
             Returns:
                  (dict): dictionary of the sum, difference, multiplication and division
             '''
             return {"sum" :x+y, "difference":x-y, "multiplication":x*y, "division":x/y}
```

- Ask for help for any function or **class**

```
In  [2]: help(simple_calculator)
```

Anonymous Functions λ

- A **Python** *lambda* function is an **anonymous function** expressed as a single statement

  lambda <arguments> : <return expression>

```
In  [1]: f = lambda x,y: {“sum” :x+y, “difference”:x-y, “multiplication”:x*y, “división”:x/y}
```

Exercise

Hit target

The height of the ball can be modeled as:

$$y(t) = -\frac{1}{2}gt^2 + v_0 t \sin\theta$$

where v_0 is the speed the ball has been thrown with, θ is the angle at which the ball has been thrown from and $g = 9.81\,\mathrm{m/s^2}$.

a)

Write a function which returns the height of the ball at a given time t.

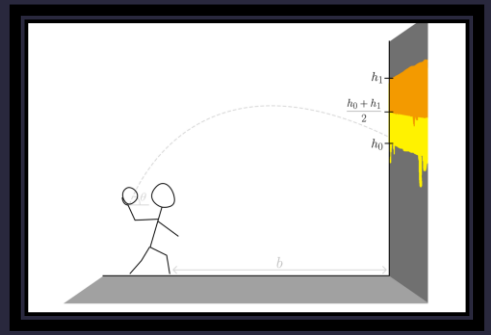
b)

One can find in our model that the ball will hit the wall at the time $T = \frac{b}{v_0 \cos\theta}$ where b is the distance between the person and the wall.

We must look at the value of $y(T)$ to be able to decide how many points the person will receive. The number of points must be calculated and returned from a function which you have to write.

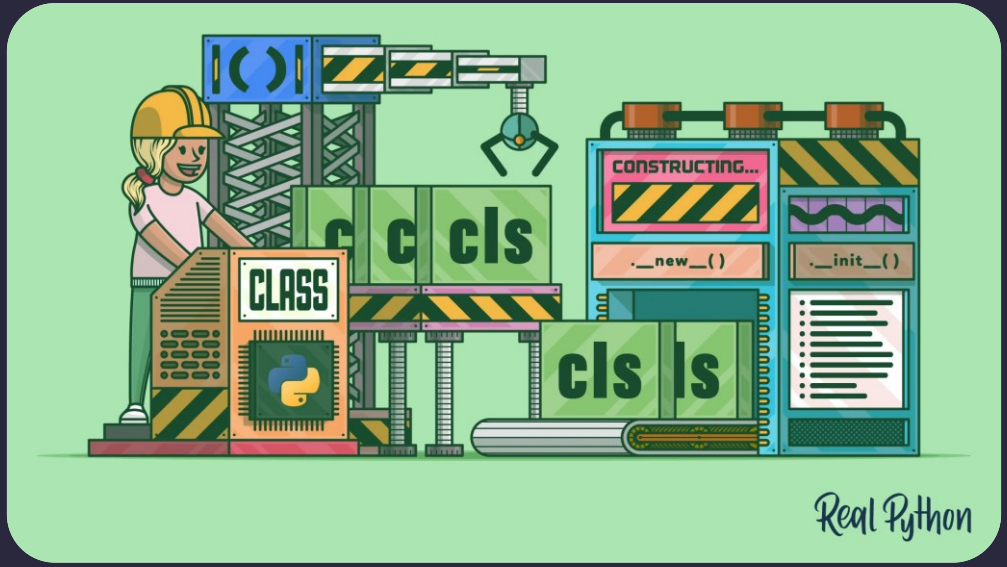
The target is painted such that it covers the wall between height h_0 and height h_1 where $h_0 < h_1$. The points are given according to the following rules:

- The person gets 0 points if $y(T) < h_0$ or $y(T) > h_1$
- The person gets 1 point if $h_0 \leq y(T) < \frac{1}{2}(h_1 + h_0)$
- The person gets 2 points if $\frac{1}{2}(h_1 + h_0) \leq y(T) < h_1$



Write a program which prints in a for loop how many points the person gets using your newly written function if $h_0 = 3\,\mathrm{m}$, $h_1 = 3.5\,\mathrm{m}$, $\theta = \frac{\pi}{4}$, $b = 3.5\,\mathrm{m}$ for $v_0 = 15, 16, 19, 22\,\mathrm{m/s}$.

/06
Classes & Objects



Classes & Objects: `class` Definition

- An **object** is a custom data structure containing both data (attributes) and functions (methods)
- **Objects** are instances of **classes**

```
In  [1]: class useless_class:
    ...:     pass
In  [2]: useless_object = useless_class()
```

- You can assign attributes or methods from **outside** the class definition

```
In  [3]: useless_object.var = “hello world”
In  [4]: useless_object.func = lambda x,y: {“sum” :x+y, “difference”:x-y, \
                                            “multiplication”:x*y,  “división”:x/y}


In  [1]: useless_object.func(1,1)
Out [2]: {'sum': 2, 'difference': 0, 'multiplication': 1, 'division': 1.0}
```

Classes & Objects: __init__

- To assign attributes and methods from **inside** the class, you need a **constructor** __init__

```
In  [1]: class useful_class:
             def __init__(self):
                 self.welcome = “hello world”
                 self.f = lambda x,y: {“sum” :x+y, “difference”:x-y, \
                                       “multiplication”:x*y,  “división”:x/y}
```

- You can pass arguments to the **constructor**

```
In  [2]: class useful_class:
             def __init__(self, name):
                 self.welcome = “hello ”+name
                 self.f = lambda x,y: {“sum” :x+y, “difference”:x-y, \
                                       “multiplication”:x*y,  “división”:x/y}

In  [3]: useful_object = useful_class()
Out [3]: TypeError: __init__() missing 1 required positional argument: 'name’

In  [4]: useful_object = useful_class(“Waleed”)
```

self argument in Python is similar to this pointer in C++

Classes & Objects: Inheritance

- **Inheritance** is creating a new class from an existing class, but with some additions or changes

```
In  [1]: class polygon:
             def __init__(self, num_of_sides):
                 self.n = num_of_sides
             def calc_area(self):
                 pass


In  [2]: class triangle(polygon):
             def __init__(self):
                 super(). __init__ (3)
             def calc_area(self, a, b, c):
                 # calculate the semi-perimeter
                 s = (a + b + c) / 2
                 area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
                 return area
```

Call parent constructor

Override `calc_area` method

Classes & Objects: Privacy

- You can hide attributes and methods to be accessed from outside the class using (__)

```
In  [1]: class useless_class:
             def __init__(self, attribute):
                 self.__hidden_attribute = attribute

             def __hidden_method(self):
                 pass


In  [2]: useless_object = useless_class()
In  [2]: useless_object.__hidden_attribute
Out [3]: AttributeError: 'useless_object' object has no attribute '__hidden_attribute'
```

- Always use ***getters*** (getter methods) to get hidden attributes

```
In  [1]: class useless_class:
             def __init__(self, attribute):
                 self.__hidden_attribute = attribute

             def get_hidden_attribute(self):
                 return self.__hidden_attribute
```

When to use classes

- Remember the zen of **Python** *“simple is better than complex”*

> Avoid overengineering datastructures. Tuples are better than objects (try namedtuple, too, though). Prefer simple fields over getter/setter functions…Built-in datatypes are your friends. Use more numbers, strings, tuples, lists, sets, dicts. Also check out the collections library, especially deque.
>
> —Guido van Rossum

- Use the simplest solution to the problem. A dictionary, list, or tuple is simpler, smaller, and faster than a module, which is usually simpler than a class.

namedtuples

- **namedtuples** are similar to dict, you can access a variable by a name

An example from physics

```
In  [1]: from collections import namedtuple
In  [2]: Graph  = namedtuple('Graph', ['X', 'Ri', 'Ro', 'y'])
In  [3]: G = Graph(X, Ri, Ro, y)
```

- X is node feature
- Ri, Ro are adjacency matrices
- y is the label vector

/07 Be Pythonista

One-liners: List comprehension

- List comprehension helps you quickly create and modify lists
- Usage: [expression + context]

```
In  [1]: a = [x for x in range(10)]

In  [2]: a = []
In  [3]: for x in range(10):
             a.append(x)
```

- List comprehension can contain if statements

```
In  [4]: customers = [('John', 240000), ('Alice', 120000), ('Anna', 1100000), ('Zach', 44000)]
In  [5]: # your high-value customers earning >$1M
In  [6]: whales = [x for x,y in customers if y>1000000]
In  [7]: whales
Out [7]: ['Anna']
```

One-liners: map() function

- **map()** function that takes as input arguments a **function object f** and a **sequence s**
- The **map()** function then applies the **function f** on each element in the **sequence s.**

➢ **Problem**: given a list of strings, your task is to create a new list of tuples, each consisting of a Boolean value and the original string. The Boolean value indicates whether the string **'anonymous'** appears in the original string

```
In  [1]: # the list of strings
In  [2]: txt = ['lambda functions are anonymous functions.',
                'anonymous functions dont have a name.',
                'functions are objects in Python.']
```

One-liners: map() function

- **map()** function that takes as input arguments a **function object f** and a **sequence s**
- The **map()** function then applies the **function f** on each element in the **sequence s**.

➢ **Problem**: given a list of strings, your task is to create a new list of tuples, each consisting of a Boolean value and the original string. The Boolean value indicates whether the string 'anonymous' appears in the original string

```
In  [1]: # the list of strings
In  [2]: txt = ['lambda functions are anonymous functions.',
                'anonymous functions dont have a name.',
                'functions are objects in Python.']

In  [3]: output = list(map(lambda s: (True, s) if 'anonymous' in s else (False, s), txt))
Out [3]: output
[(True, 'lambda functions are anonymous functions.'),
 (True, 'anonymous functions dont have a name.'),
 (False, 'functions are objects in Python.')]
```

Decorators:

- A **decorator** is a function that takes one function as input and returns another function
- Function inside function (**inner functions**) is perfectly normal in **Python**

```
In  [1]: def add_numbers(x,y):
             return x+y
```

- Modify the behaviour of this functions without modifying the code (decorate it)

```
In  [2]: def square_it(f):
             def new_func(*args, **kwargs):
                 result = f(*args, **kwargs)
                 return result**2
             return new_func

In  [3]: @square
         def add_numbers(x,y):
             return x+y

In  [4]: add_numbers(2,2)
Out [4]: 16
```

Questions:

1. Write a program to count Even and Odd numbers in a list using **lambda**
   e.g., `list1 = [10, 21, 4, 45, 66, 93, 1]`

2. Write a program to create a recursive function to calculate the sum of numbers from 0 to 10.

3. How to flatten all sublists of a list, no matter how deeply nested using **Python** ?
   e.g., `lol = [1, 2, [3,4,5], [6,[7,8,9], []]]`

Resources:

1. Introducing Python: Modern Computing in Simple Packages, Bill Lubanovic
2. Python One-liners: Write Concise, Eloquent Python Like A Professional, Christian Mayer
3. https://realpython.com/
4. Programming exercises with applications in physics, Morten Hjorth-Jensen

/THANKS!

/DO YOU HAVE ANY QUESTIONS?

w.esmail@fz-juelich.de
waleed.physics@gmail.com

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon**, and infographics & images by **Freepik**