Introduction to Python

Gloria Beraldo (gloria.beraldo@unipd.it)
Department of Information Engineering, University of Padova

Topics:

- History
- Philosophy of Python
- Get started with Python
- Modules in Python

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

History

Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office (a government-run research lab in Amsterdam) would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).

python™

Python was conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC programming language.

Foreword for "Programming Python" (1st ed.)
https://www.python.org/doc/essays/foreword/

History

One core of Python is represented by the aphorism: Simple is better than complex

*As it turned out, Python is **remarkably free from many of the hang-ups** of conventional programming languages.*

*First, the **use of indentation reduces visual clutter** and makes programs shorter, thus reducing the attention span needed to take in a basic unit of code. Second, it **allows the programmer less freedom in formatting**, thereby enabling a more uniform style, which makes it easier to read someone else's code.*

*As an object-oriented language, Python aims to encourage the creation of **reusable code**.*

*I will gladly admit that **Python is not the fastest** running scripting language. It is a good runner-up though.*

*In addition, many consider **using Python a pleasure**.*

Foreword for "Programming Python" (1st ed.)

https://www.python.org/doc/essays/foreword/

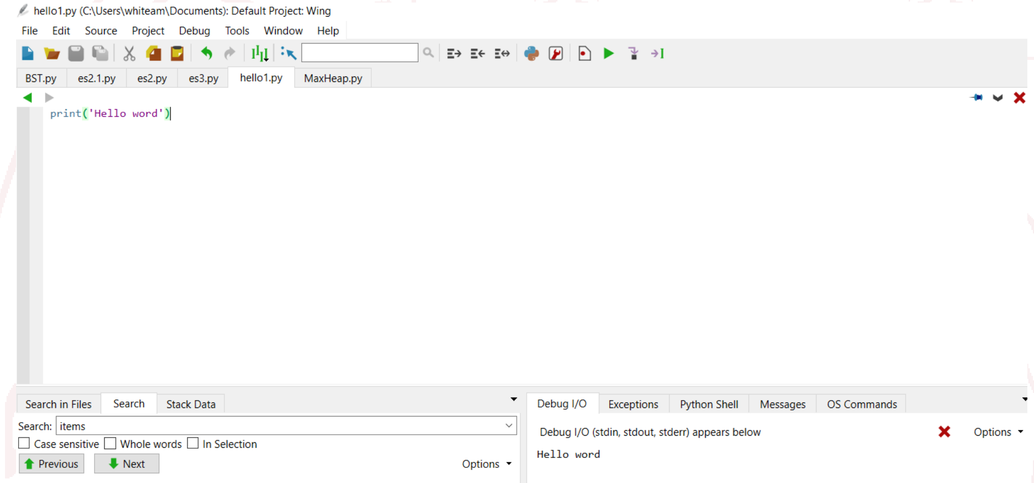
What is python?

- Multi-paradigm programming language

- Object-oriented programming

- Interpreted language.

  The commands are executed through a software called interpreter. The interpreter takes the commands, evaluates them, and returns the result of the command. The interpreter can be used interactively from the command line or a series of instructions can be saved in a file known as source code or script with a .py extension (e.g., hello.py)



- High-level language

Strengths of Python

- Simple to learn and use
- Powerful and fruitful
- Open source (https://www.python.org/)
- Easily integrable with C/C++ and Java

Python is used by Google, IBM, Facebook, NASA, Industrial Light & Magic (which created special effects for Star Wars),

https://www.python.org/about/success/

Python: *Simple is better than complex*

Java:

```
class HelloWorldApp {
    public static void main(String[] args) {
    System.out.println(“Hello, World!”);
    }
}
```

Total characters: 87
(Without “Hello, world!’’)

Python:

```
print(‘Hello, world!’)
```

Total characters : 7

No { }
No ;

Python: Identation

```
x=3
if x > 10:
 print('Greater than 10')
else:
 print ('Otherwise')
```

Search in Files | Search | Stack Data

arch: items

Case sensitive | Whole words | In Selection

Previous | Next | Options

Debug I/O | Exceptions | Python Shell | Messages | OS Commands

Debug I/O (stdin, stdout, stderr) appears below

```
Otherwise
```

```
x=3
if x > 10:
print('Greater than 10')
else:
 print ('Otherwise')
```

Search in Files | Search | Stack Data

arch: items

Case sensitive | Whole words | In Selection

Previous | Next | Options

Debug I/O | Exceptions | Python Shell | Messages | OS Commands

Ignore this exception location | Option:

File "C:\Users\whiteam\Documents\hello1.py", line 3, in ?

print('Greater than 10')

builtins.IndentationError: expected an indented block (hello1.py, line 3)

Python uses **indentation**

35 keywords in Python

Python has these 35 keywords or reserved words that they can't be used as identifiers.

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

Built-in types of data in Python

The default built-in types of data in Python are the following:

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

NB: char does not exist, it is a 1-length string

You can get the data type by using the **type()** function

```
x = 5
print(type(x))
```

```
<class 'int'>
```

https://www.w3schools.com/python/trypython.asp?filename=demo_type

Built-in types of data in Python

Examples:

- int: 6, 3, 10, −2, etc.
- float: 6.0, −3.2, 4.5E10
- Complex: 1+ 3j
- str: ‘hi there’, “how are you”
- list: [], [6, ‘hi there’]
- dict: {}, {‘hi there’:6, ‘how are you’:4}
- bool: True, False

A dictionary is a mapping of keys to values.

True corresponds to 1, False to 0.

Variables in Python

- Variable names consist of letters, numbers and underscores
- The first character cannot be a number or a symbol:

  The following are therefore valid: “x”, “hello”, “x13”, “x1_y”, “_”, “_hello12”
  The following are not valid: “1x”, “x-y”, “$a” , “$b”
- Must not be declared (dynamic typing)
- Cannot be used before a value is assigned to them
- Python is **not a strictly typed language** (or at least try not to be).That is, most of the time the type of object assigned to a variable is **established in such a way automatic**.
- The declaration of a new variable therefore always comes from an operation assignment: **=**
- The assignment statement = creates the variable and allows it to be saved inside it a value.

  Examples
  x=5
  name = ‘Marco’
  start, finish =2,100

Input function in Python

The **input function** allows the user to specify a from the keyboard value and save it in the specified variable:

```
n = input('Enter a value')

print(n)
```

Therefore, the output depends on what the user types:

Debug I/O | Exceptions | Python Shell | Mess

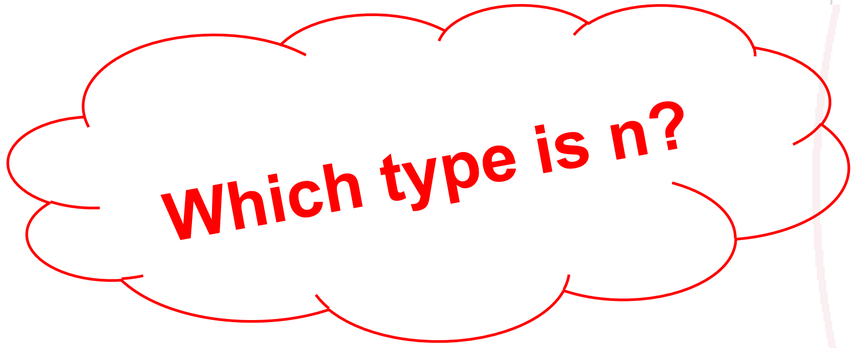
Debug I/O (stdin, stdout, stderr) appears below

```
Enter a value3
3
```

Debug I/O | Exceptions | Python Shell | Mess

Debug I/O (stdin, stdout, stderr) appears below

```
Enter a value'ciao'
'ciao'
```



<class 'str'>

eval() function in Python

The **eval function** allows Python to automatically parse the type of the variable based on its use

```
n = input('Enter a value')
n = eval(n)
n-1
print(type(n))
```

Debug I/O (stdin, stdout, stderr) appears below

```
Enter a value2
<class 'int'>
```

Debug I/O (stdin, stdout, stderr) appears below

```
Enter a value2.0
<class 'float'>
```

Casting in Python

Most of the time we already have an object and want to create another object using one or more existing objects.

```
y = '6'      # String '6' is our object,
             # y is an object which points at '6'
x = int(y)   # x now is an int object which point at the object y
print(x)     # this print the int 6
```

EXPLICIT TYPECAST

variable = target_type (source_object)

```
z = float('6.3')   # crea una variabile z di tipo float = 6.3
w = str(z)         # assegna a w la una stringa "6.3"
u = list(w)        # crea una lista ['6','.','3']
```

Variable environment

A notable difference between Python and other languages is the definition of the environment of the variables.

The validity of variable environment is enough extended.

For example, pay attention to:

Python maintains the same environment for conditional and looping constructs

A)

```
v = True
if v is True:
   phone = 5552368
print(phone)
```

We enter in the loop and so there Is no error

B)

```
v = False
if v is True:
   phone2 = 5552368
print(phone2)
```

----> 4 print(phone2)

NameError: name 'phone2' is not defined

Variable environment

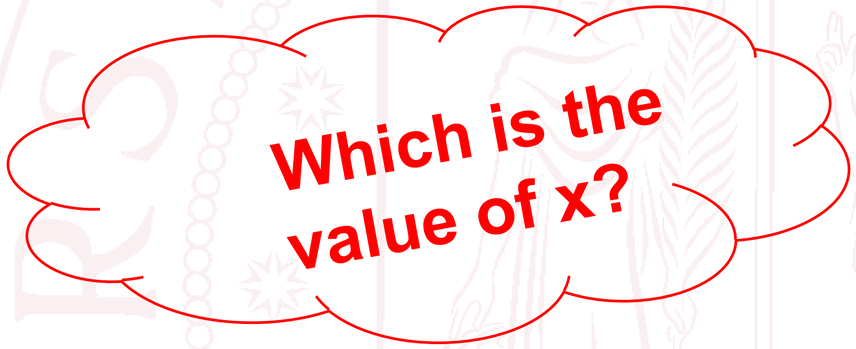
Differently from what we saw for the variables in the cycles and in the decision constructs, the validity of the variables declared inside a function is restricted to function block only and all variables defined therein are deleted outside.

These are therefore called "**LOCAL variables**"

In python a function is defined using def

```
x = 0 # Global variable
def my_function():
    x = 123 # Local variable inside my_function


my_function()
print(x)
```




```
Debug I/O | Exceptions | Python Shell | M

Debug I/O (stdin, stdout, stderr) appears below

0
```

Variable environment

To keep the value assigned to a variable inside a function, it is necessary to define the variable as global.

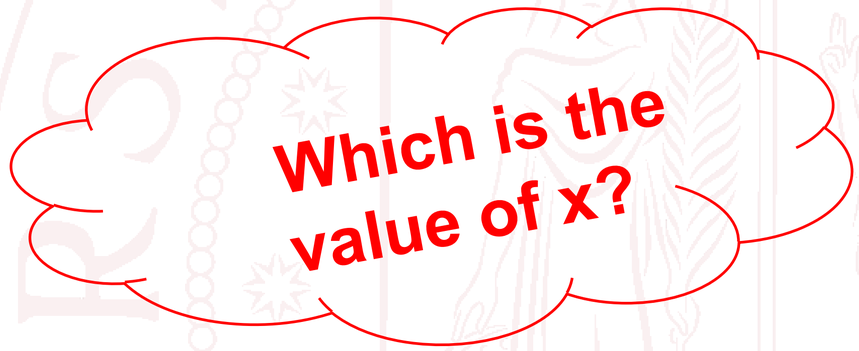
```
x = 0 # Global variable
def my_function():
    global x # Declaration that x is a global variable
    x = 123 # Access to the gloabal variable by assignement

my_function()
print(x)
```

```
Debug I/O | Exceptions | Python Shell | Mess

Debug I/O (stdin, stdout, stderr) appears below

123
```



Packing and Unpacking of Sequences in Python

PACKING:

```
data = 2, 4, 6, 8
print(data)
```

Output

Debug I/O (stdin, stdout, stderr) appears below

```
(2, 4, 6, 8)
```

UNPACKING:

```
a, b, c, d = range(7, 11)
print(a)
print(b)
print(c)
print(d)
```

Debug I/O (stdin, stdout, stderr) appears below

```
7
8
9
10
```

MULTIPLE ASSIGNMENT:

```
a, b, c = 6, 2, 5

print(a)

print(b)

print(c)
```

Debug I/O (stdin, stdout, stderr) appears below

```
6
2
5
```

Lists and Tuples in Python

Both lists and tuples are used for storing objects in python (i.e., container).

Objects that are stored in lists and tuples can be of any type.

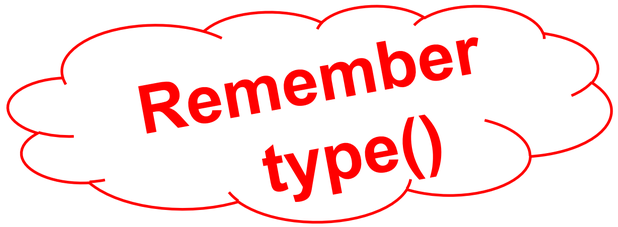
FEATURES OF A LIST:

- Lists are one of the most flexible and powerful containers in Python,
- You can use Python lists to store data of **multiple types** simultaneously,
- Lists help preserve data sequences and further process those sequences in other ways,
- Lists are dynamic → **easily to modify**,
- Lists are **mutable**,
- Lists are ordered,
- An index is used to traverse a list,
- Iterations are time-consuming,
- Lists consume more memory.

FEATURES OF A TUPLE:

- Tuples are used to store heterogeneous and homogeneous data,
- Tuples are **immutable** in nature → **faster than list**,
- Tuples are ordered,
- An index is used to traverse a tuple,
- Tuples are similar to lists. It also preserves the **data sequence**.
- Iterations are faster than list
- Tuples consume less memory than lists.

Lists and Tuples in Python: Syntax



LIST SYNTAX:

- A list is initiated with the **[ ]**

```
num_list = [1,2,3,4,5]

alphabets_list = ['a','b','c','d','e']
```

- A list can contain data of different data types.

```
mixed_list = ['a', 1,'b',2,'c',3,'4']
```

- You can create nested lists as well. A nested list is a list inside a list.

```
nested_list = [1,2,3,[4,5,6],7,8]
```

TUPLE SYNTAX:

- A tuple is initiated with the **()**

```
num_tuple = (1,2,3,4,5)

alphabets_tuple = ('a','b','c','d','e')
```

- A tuple can contain data of different data types.

```
mixed_tuple = ('a', 1,'b',2,'c',3,'4')
```

- You can create nested tuples as well. A nested tuple is a tuple inside a tuple.

```
nested_tuple = (1,2,3,(4,5,6),7,8)
```

```
print(len(nested_list))
```

```
print(len(nested_tuple))
```

NB: the length is 6 in both examples

Lists and Tuples in Python: What is the difference?

The primary difference between tuples and lists is that **tuples are immutable** as opposed to **lists which are mutable**. Therefore, it is possible to change a list but not a tuple.

The **contents of a tuple cannot change** once they have been created in Python due to the immutability of tuples.

The **length** of tuples is also **fixed**. They remain the same length throughout the lifecycle of the program.

LIST

```
names = ["Raj","John","Jabby","Raja"]
print(names[2])
```

Debug I/O (stdin, stdout, stderr) appears below

```
Jabby
```

```
names[2] = "Kelly"

print(names[2])
```

Debug I/O (stdin, stdout, stderr) appears below

```
Kelly
```

TUPLE

```
names = ("Raj","John","Jabby","Raja")

print(names[2])
```

```
names[2] = "Kelly"
```

builtins.TypeError: 'tuple' object does not support item assignment

Operations with Lists in Python

As the list is mutable, it has **many inbuilt operations** that you can use for modifying it.

ACCESS TO ELEMENTS IN A LIST L

Access to an element in the list:

L[index]

```
L = [1,2,3,4]

print(L[2])
```

```
Debug I/O (stdin, stdout, stderr) appears below

3
```

Access to the last element in the list:

L[-1]

```
print(L[-1])
```

```
Debug I/O (stdin, stdout, stderr) appears below

4
```

Access to a set of elements in the list:

L[index1: index2] where index1 is included, index2 is excluded

```
print(L[1:3])
```

```
Debug I/O (stdin, stdout, stderr) appears below

[2, 3]
```

Operations with Lists in Python

As the list is mutable, it has **many inbuilt operations** that you can use for modifying it.

ADD AN ELEMENT IN A LIST L

L.append(element)

```
L=[]
L.append(2)
L.append(3)
print(L)
```

Debug I/O (stdin, stdout, stderr) appears below

```
[2, 3]
```

ADD MULTIPLE ELEMENTS IN A LIST L

L.extend([element1, element2, … , element_n])

```
L=[]
L.extend([1,2,3])
print(L)
```

Debug I/O (stdin, stdout, stderr) appears below

```
[1, 2, 3]
```

ADD AN ELEMENT IN A SPECIFIC POSITION IN A LIST L

L.insert([position, element])

```
L = [1,2,3,4]
L.insert(1,5)
print(L)
```

Debug I/O (stdin, stdout, stderr) appears below

```
[1, 5, 2, 3, 4]
```

Operations with Lists in Python

As the list is mutable, it has **many inbuilt operations** that you can use for modifying it.

REVERSE A LIST L

L.reverse()

```
L = [5,2,3,4]
L.reverse()
print(L)
```

Debug I/O (stdin, stdout, stderr) appears below

```
[4, 3, 2, 5]
```

SORT A LIST L

L.sort()

```
L = [5,2,3,4]
L.sort()
print(L)
```

Debug I/O (stdin, stdout, stderr) appears below

```
[2, 3, 4, 5]
```

VERIFY IF AN ELEMENT IS/NOT BE IN A LIST L

element in L

element not in L

```
L = [5,2,3,4]
print(2 in L)
print(4 not in L)
```

Debug I/O (stdin, stdout, stderr) appears below

```
True
False
```

Operations with Lists in Python

As the list is mutable, it has **many inbuilt operations** that you can use for modifying it.

REMOVE AN ELEMENT IN A LIST L

L.remove(element)

```
L = [5,2,3,4]
L.remove(2)
print(L)
```

```
Debug I/O (stdin, stdout, stderr) appears below
[5, 3, 4]
```

REMOVE THE LAST ELEMENT IN A LIST L

L.pop()

```
L = [5,2,3,4]
L.pop()
print(L)
```

```
Debug I/O (stdin, stdout, stderr) appears below
[5, 2, 3]
```

FIND THE INDEX OF AN ELEMENT IN A LIST L

L.index(element)

```
L = [5,2,3,4]
print(L.index(3))
```

```
Debug I/O (stdin, stdout, stderr) appears below
2
```

Dictionaries in Python

Dictionaries store pairs of data and keys.

Dictionaries **do not allow duplicates**.

D = { key1: value1, key2: value2, …., key_n: value_n}

```
D = {10000: "Mario Rossi", 10001: "Maria Bianchi", 10002: "Pinco Pallino"}
```

```
empty_dict = {}
```

ACCESS DATA USING KEY IN A DICTIONARY D
D[key]

```
print(D[10000])
```

Debug I/O (stdin, stdout, stderr) appears below

```
Mario Rossi
```

REMOVE DATA USING KEY IN A DICTIONARY D
del D[key]

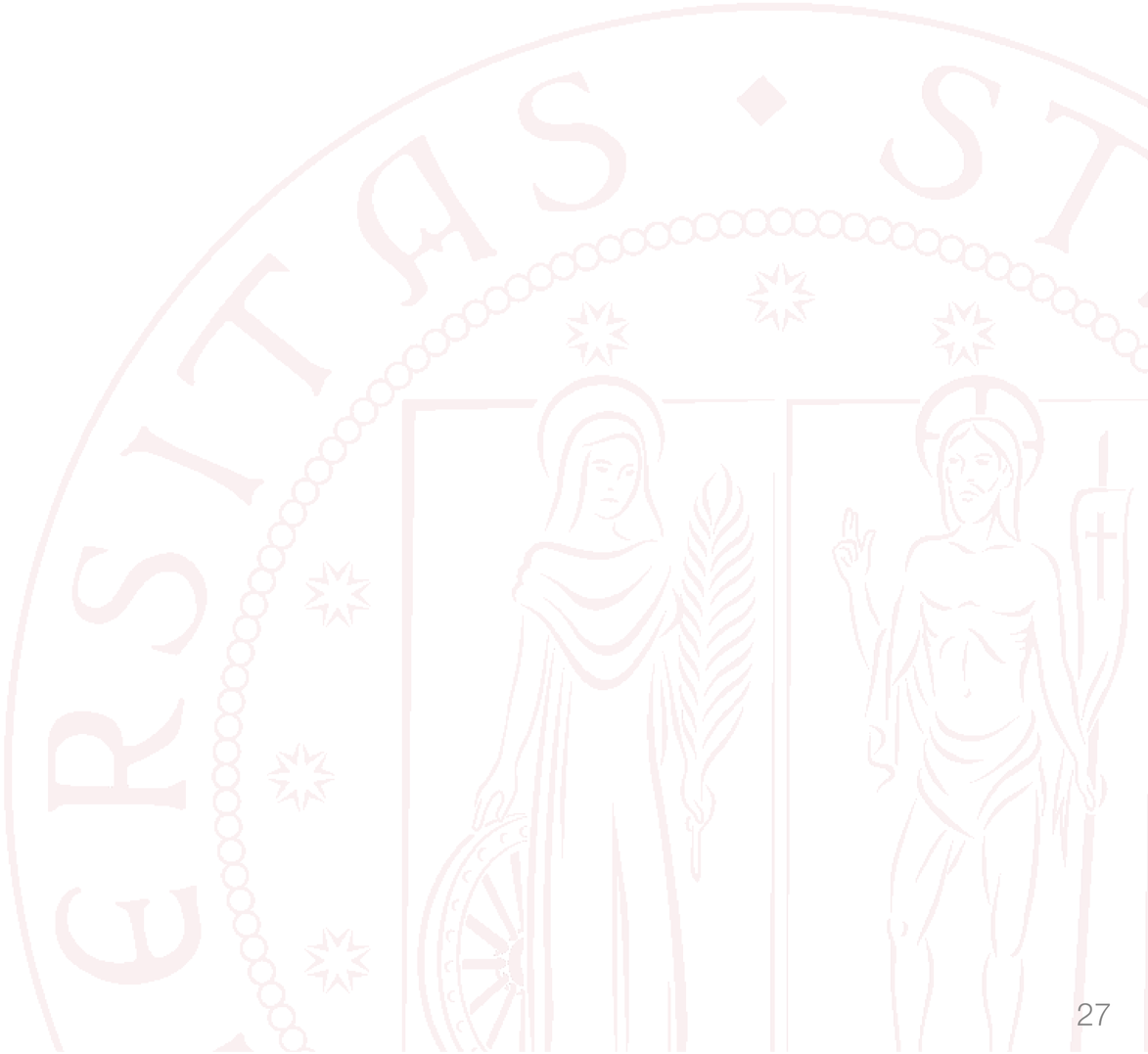
```
D = {10000: "Mario Rossi", 10001: "Maria Bianchi", 10002: "Pinco Pallino"}
del D[10000]
print(D)
```

Debug I/O (stdin, stdout, stderr) appears below

```
{10001: 'Maria Bianchi', 10002: 'Pinco Pallino'}
```

Mathematical Operations in Python

- Sum a+b
- Subtraction a-b
- Multiplication a*b
- Division a/b
- Module a%b
- Raised to the power a**b



Logic Operations in Python

- And a and b
- Or a or b
- Not not a

The **precedence** of the logical operator from the highest to lowest: **not, and, or**.

```
a = True
b = False

print(a and b)
print(a or b)
print(not a)
print(not b)
print(not a and b or a)
```

Debug I/O (stdin, stdout, stderr) appears below

```
False
True
False
True
True
```

Operations with String in Python

- Concatenation of string a + b
- Convert to uppercase a.upper()
- Convert to lowercase a.lower()

```
a="Hello"
b = "World"

print(a + " " + b)
print(a.upper())
print(b.lower())
```

Debug I/O (stdin, stdout, stderr) appears below

```
Hello World
HELLO
world
```

Loops in Python

WHILE LOOP

The while loop repeats the block of indented statements until a certain condition is true

```
a = 0
while a < 10:
    a = a + 1
print(a)
```

Debug I/O (stdin, stdout, stderr) appears below

```
10
```

FOR LOOP

The for loop iterates over iterable objects (lists, strings, ranges,dictionaries, etc. etc.):

```
for a in range(0,10):
    print(a)
```

Debug I/O (stdin, stdout, stderr) appears below

```
0
1
2
3
4
5
6
7
8
9
```

Keep attention to identation

Functions in Python

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.
- A function can return data as a result.

```
def my_function(food):
  for x in food:
    print(x)
  food.pop()


fruits = ["apple", "banana", "cherry"]

my_function(fruits)
print(fruits)
```

Debug I/O (stdin, stdout, stderr) appears below

```
apple
banana
cherry
['apple', 'banana']
```

NB: The list is changed

```
def sum(a, b):
    return a + b


c = sum(3,4)
print(c)
```

Debug I/O (stdin, stdout, stderr) appears below

```
7
```

Importing modules in Python

When you write a program in Python, you will be using code written by someone else.

Code written by others is usually provided in a **module**.

To use a module needs to be imported. For instance:

```
import math, random, csv
import itertools, functools
import numpy
```

To import **only part** of the module:

```
from nxviz import ArcPlot
```

To import a module and rename it:

```
import matplotlib.pyplot as plt
```

Create a new module in Python

To create a new module in python simply create a new file with the .py extension

Before executing code, Python interpreter reads source file and define few special variables/global variables.

```
print ("Always executed")

if __name__ == "__main__":
    print ("Executed when invoked directly")
else:
    print ("Executed when imported")
```

If the python interpreter is running that module (the source file) **as the main program**, it sets the special __name__ variable to have a value “__main__”.

test.py

```
print ("Always executed")

if __name__ == "__main__":
    print ("Executed when invoked directly")
else:
    print ("Executed when imported")
```

running
python test.py

```
Debug I/O (stdin, stdout, stderr) appears below

Always executed
Executed when invoked directly
```

Create a new module in Python

If this file is **being imported from another module**, __name__ will be set to the module's name.
Module's name is available as value to __name__ global variable.

test.py

```
print ("Always executed")

if __name__ == "__main__":
    print ("Executed when invoked directly")
else:
    print ("Executed when imported")
```

test1.py

```
import test
```

running
python test1.py

Debug I/O (stdin, stdout, stderr) appears below

```
Always executed
Executed when imported
```

Questions