Python Basics

Python

Outline

- About python
- Program compilation
- Installation
- Syntax
- Data types (1)
- Arithmetic operations
- Data types (2)
- Logic and execution flow
- Functions
- Classes

About Python

History

Python is a general-purpose programming language first introduced in 1991 by Guido van Rossum .

With python 2.0 being released in 2000 , and python 3.0 released in 2008 .

It is a high-level and dynamically typed language with garbage-collection feature .

Program compilation

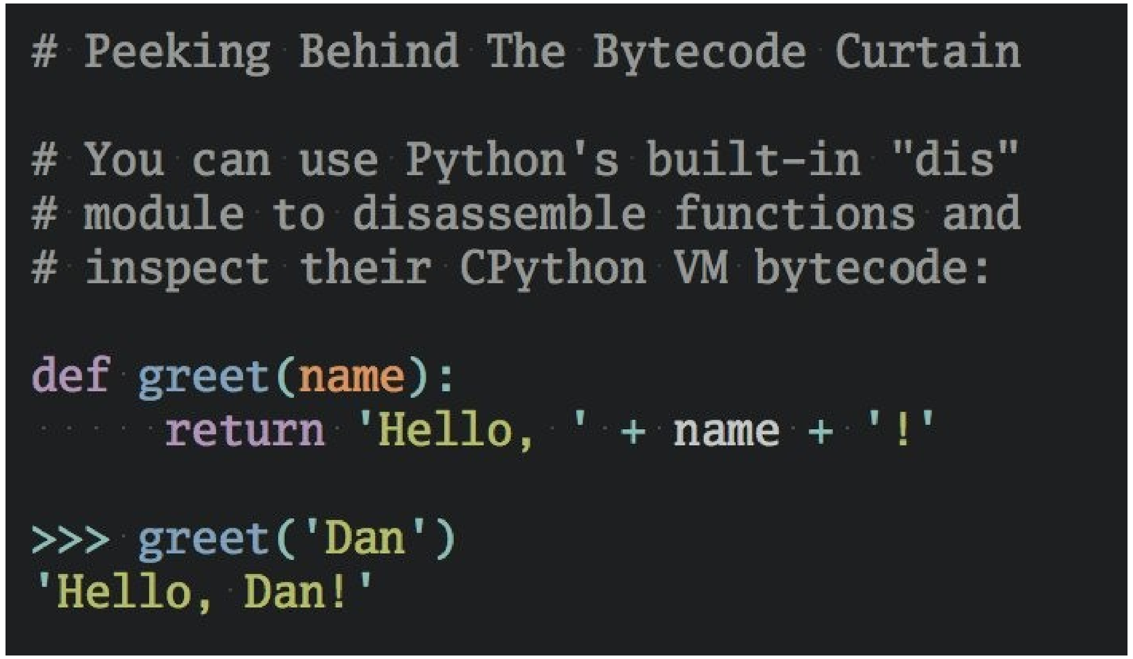
Python bytecode and P.V.M.

Python code is first compiled to python bytecode, a lower level language that python's virtual machine uses to execute each line of code typed in the source code, the PVM is installed by the python installer and runs on top of the operating system (Windows, MacOS, etc..) .

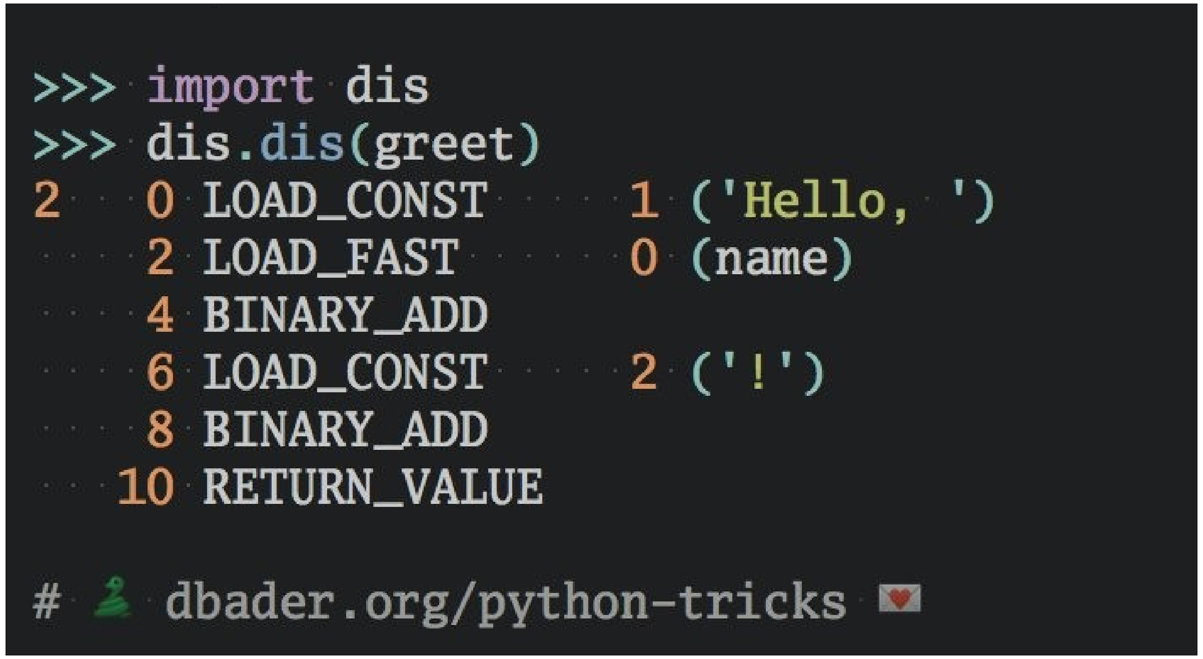
The bytecode when cached is stored in .pyc & .pyo files and if no change made in the source code file, python skips the compilation to bytecode process and uses the old cached bytecode file .

Program compilation

Python bytecode and P.V.M.



```
# Peeking Behind The Bytecode Curtain

# You can use Python's built-in "dis"
# module to disassemble functions and
# inspect their CPython VM bytecode:

def greet(name):
    return 'Hello, ' + name + '!'

>>> greet('Dan')
'Hello, Dan!'
```



```
>>> import dis
>>> dis.dis(greet)
2     0 LOAD_CONST     1 ('Hello, ')
      2 LOAD_FAST      0 (name)
      4 BINARY_ADD
      6 LOAD_CONST     2 ('!')
      8 BINARY_ADD
     10 RETURN_VALUE

# 🐍 dbader.org/python-tricks 💌
```

By dbader

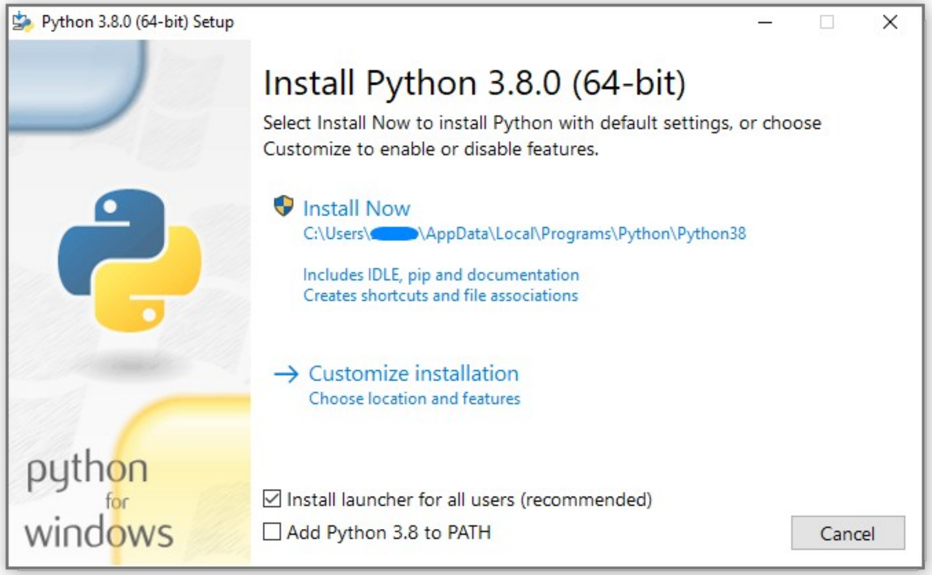
Grow Through

Installation

Python & IDLE

Python can be downloaded from the official website www.python.org .It comes with IDLE which is a simple python text editor used to type and edit python files .

We will be using python 3.8.x in this workshop .



Python 3.8.0 (64-bit) Setup

Install Python 3.8.0 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\[redacted]\AppData\Local\Programs\Python\Python38

Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

python for windows

☑ Install launcher for all users (recommended)
☐ Add Python 3.8 to PATH

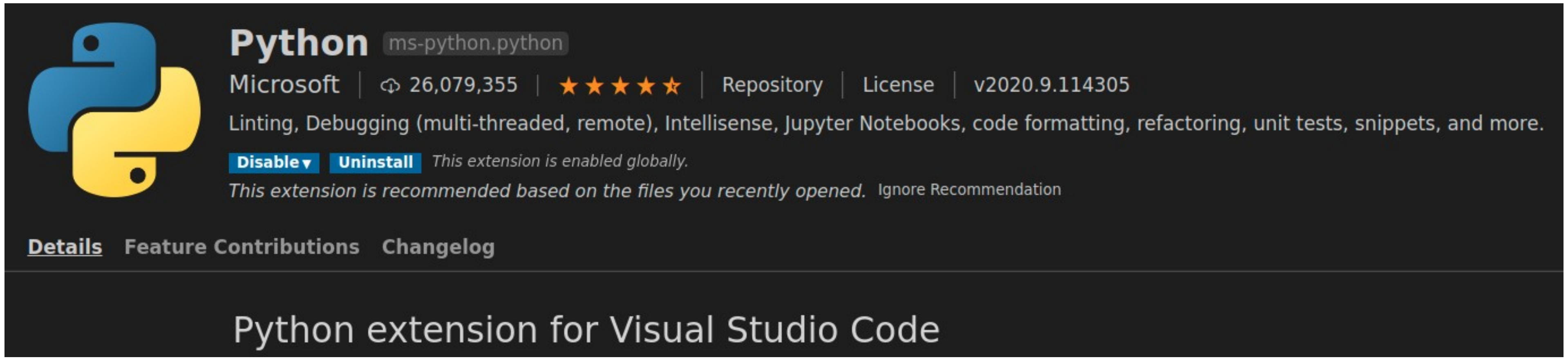
Cancel

Grow Through

Installation

IDE : VS-Code

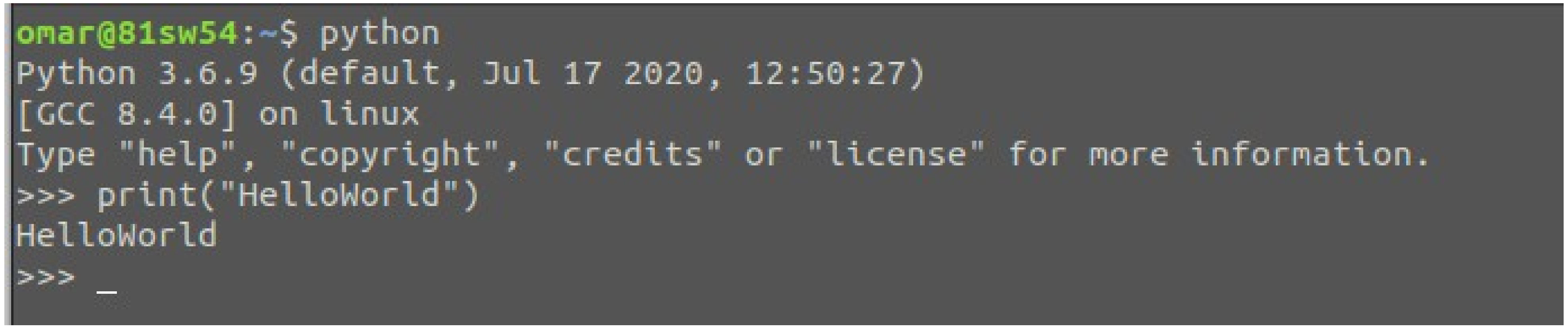
A very popular and open-source IDE to use is visual studio code, it has a python extension made and maintained by Microsoft makes writing python code an easy task .


Python ms-python.python
Microsoft | 26,079,355 | ★★★★☆ | Repository | License | v2020.9.114305
Linting, Debugging (multi-threaded, remote), Intellisense, Jupyter Notebooks, code formatting, refactoring, unit tests, snippets, and more.
Disable▾ Uninstall *This extension is enabled globally.*
This extension is recommended based on the files you recently opened. Ignore Recommendation
Details Feature Contributions Changelog
Python extension for Visual Studio Code

Installation

Running a python program

Use python interpreter to run simple python code, python interpreter can be opened using python.exe program or calling "python" on any of the command line tools if python is already added to the PATH .



```
omar@81sw54:~$ python
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("HelloWorld")
HelloWorld
>>> _
```

Syntax

Variables

variables are the names of an object in python .When we try to call an object we use the name referring to it .

```
>>> x = 4
>>> type(x)
<class 'int'>
>>> y = 4.0
>>>type(y)
<class 'float'>
>>>print(x, y)
4 4.0
```

Syntax

Comments

#

There is no mutli-line comment in python . :(

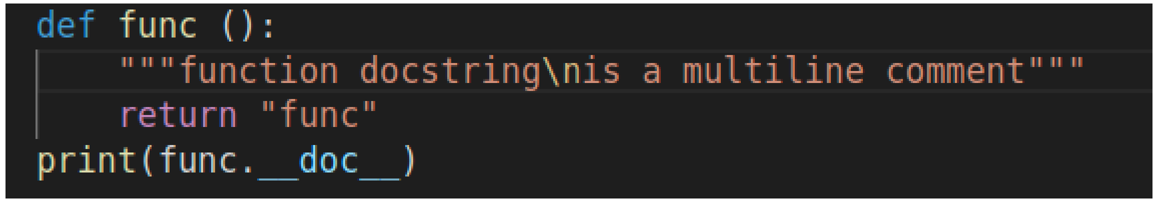
Syntax

Docstrings

Docstring is a commented string that is used to document objects in python .

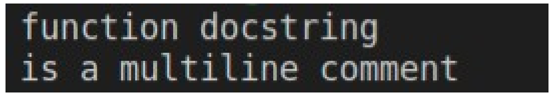
It’s used by inserting the comment inside a triple double or single quotes .

Docstrings and comments aren’t so much alike, docstring doesn’t get ignored by the compiler like regular comments but is stored inside the program can can be viewed in program's life time .



```python
def func ():
    """function docstring\nis a multiline comment"""
    return "func"
print(func.__doc__)
```

Outputs



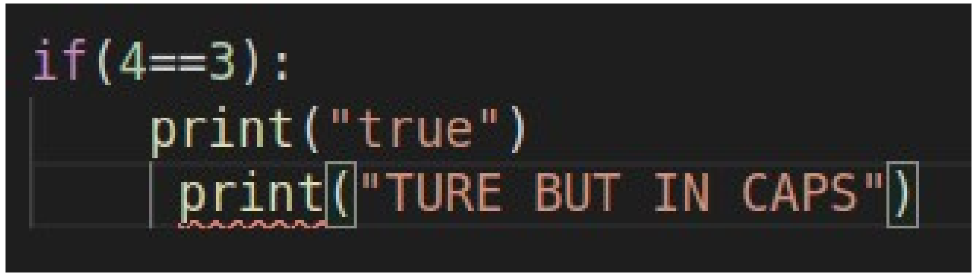
```
function docstring
is a multiline comment
```

Syntax

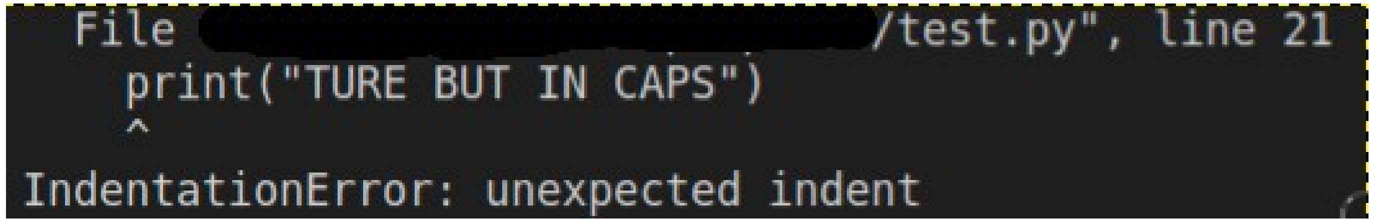
Indentation

Indentation is a way in python to determine the scope of objects, like the curly braces in other programming languages .

We always indent a block of code one space or more to represent that it's inside another block and all the lines of the inner block must be indented the same amount of spaces, otherwise this causes an indentation error .



```
if(4==3):
    print("true")
     print("TURE BUT IN CAPS")
```



```
  File [redacted]/test.py", line 21
    print("TURE BUT IN CAPS")
    ^
IndentationError: unexpected indent
```

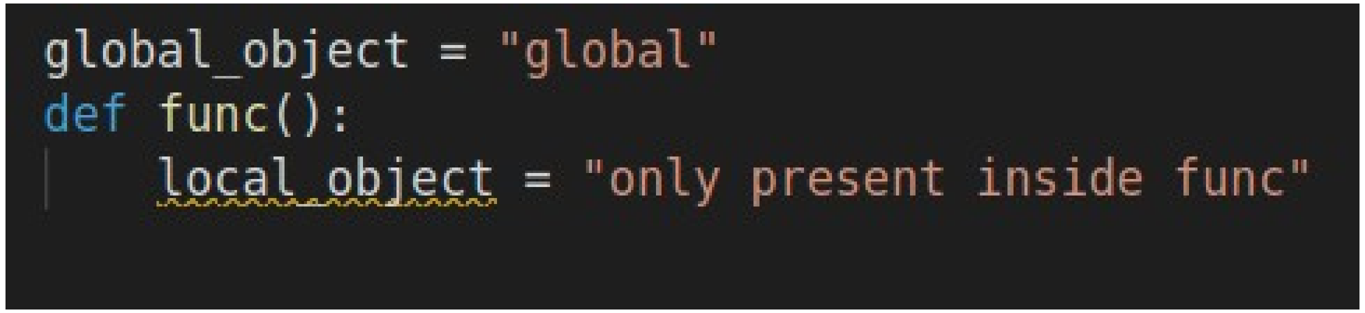
Grow Through

Syntax

Scopes

A scope of an object are the places on the code that the object can be called in and be recognized by the interpreter .

An object generally has the scope of its indentation level, for zero indentation level the object would have a global scope, if the object is defined inside a function then it will be available inside its function only (will have its function's inner scope) .



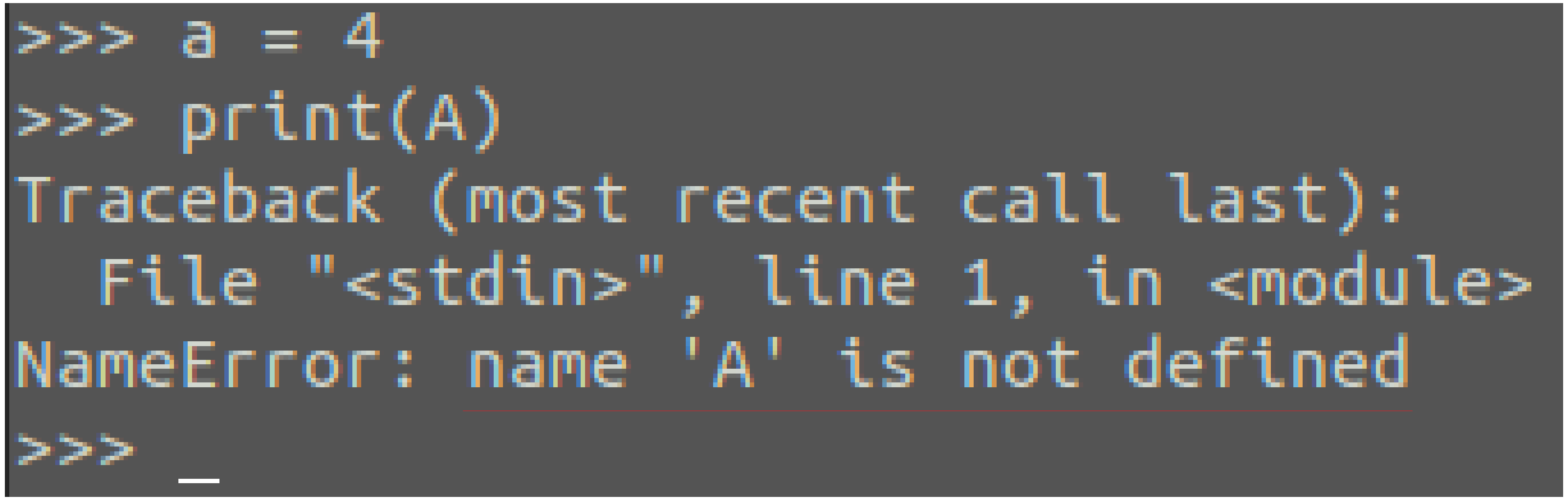
```
global_object = "global"
def func():
    local_object = "only present inside func"
```

Grow Through

Syntax

Case sensitivity

Python is a case-sensitive language .

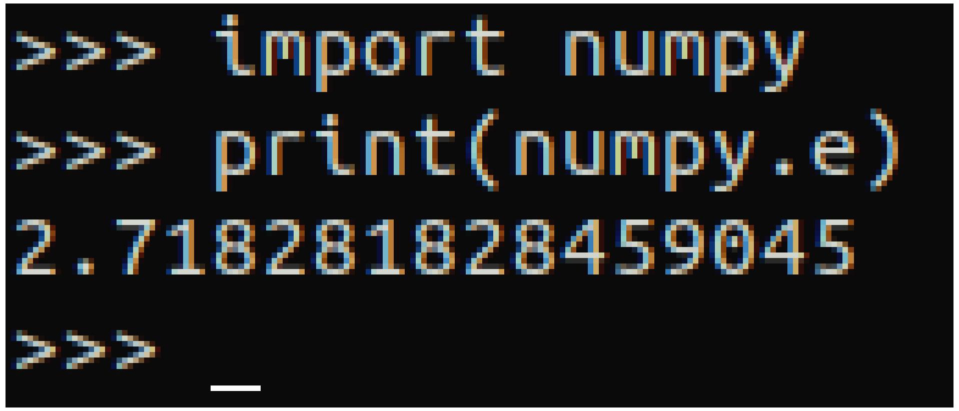


```
>>> a = 4
>>> print(A)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'A' is not defined
>>> _
```

Syntax

Import Statement

“import” keyword is used to include a python package, module, class, function, etc …
into the current python project so these objects can be called inside the source code .

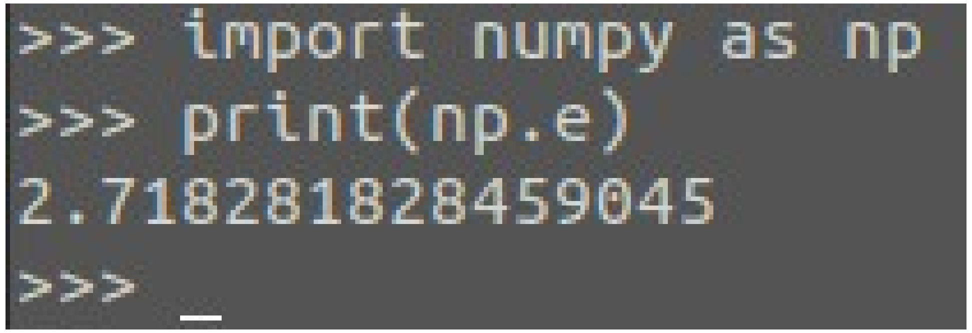


```
>>> import numpy
>>> print(numpy.e)
2.718281828459045
>>> _
```

Syntax

Import Statement

"import" keyword can be used in many different forms, one of them is "import _ as _" .

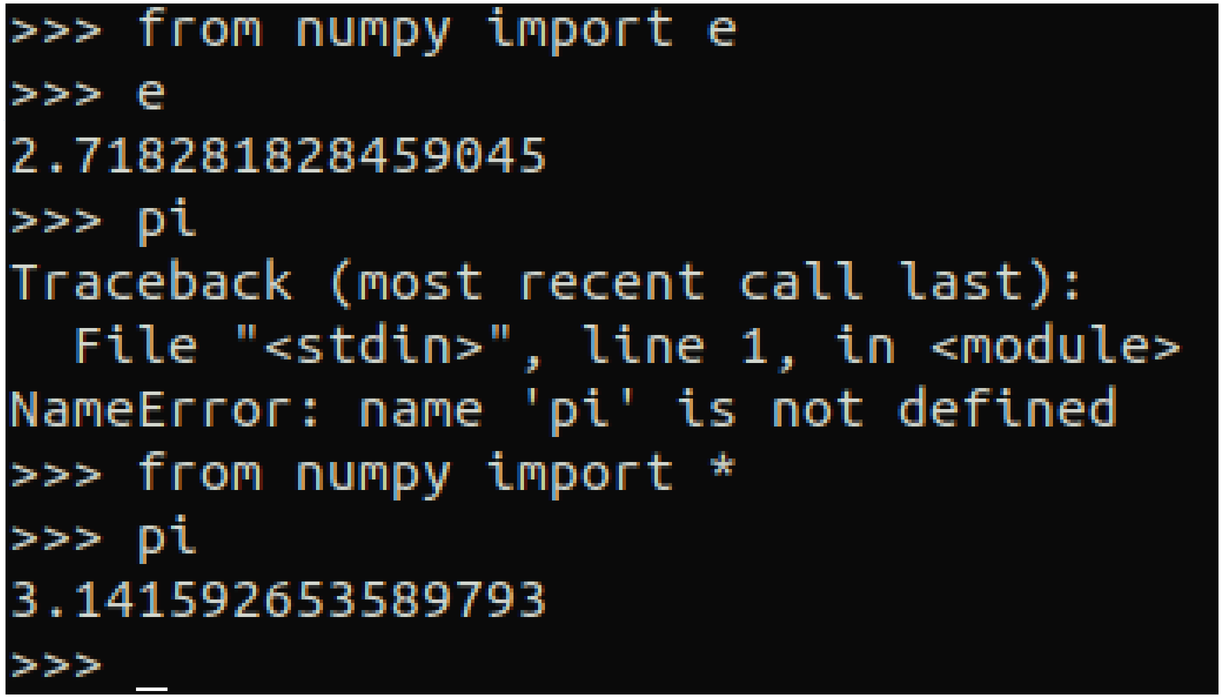


```
>>> import numpy as np
>>> print(np.e)
2.718281828459045
>>> _
```

Grow Through

Syntax

Import Statement



```
>>> from numpy import e
>>> e
2.718281828459045
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
>>> from numpy import *
>>> pi
3.141592653589793
>>> _
```

e was detected

but pi wasn't :(

* means all, so it basically tells python to import the whole module and attach it to the current namespace .

Grow Through

Syntax

Search path for import

An import statement will search for the mentioned module in the following order :

- current folder
- echo $PYTHONPATH
- sys.path

Data types -1

integer

int is a data type that stores integers, its initial size is 24 bytes and can scale up to as much memory as the machine can give . Also we don’t need to specify a data type for the variable we store our object in and we can change the variable’s data type later on the code due to python being dynamically-typed language .

```
>>>x = 4

>>>type(x)

<class 'int'>

>>>import sys

>>>sys.getsizeof(x)

28
```

Data types -1

float

float is a date type that stores floating point numbers, i.e. any real number to a limited precision .

```
>>>x = 4.0
>>>type(x)
<class 'float'>
>>>import sys
>>>sys.getsizeof(x)
24
```

Grow Through

Data types -1

string

A data type used to store a sequence of characters . A string can be initialized using single or double quotes .

```
>>>x = 'my string'
>>>type(x)
<class 'str'>
>>>import sys
>>>sys.getsizeof(x)
57
```

Data types -1

casting

Casting is the process of converting one data type to another .

```
>>>x = int(54.34)
>>>print(x)
54
>>>type(x)
<class 'int'>
```

```
>>>x = float(45)
>>>print(x)
45.0
>>>type(x)
<class 'float'>
```

Data types -1

casting

Casting can also be done with strings .

```
>>>x = int(“54”)
>>>print(x)
54
>>>type(x)
<class 'int'>
```

```
>>>x = float(“45.34”)
>>>print(x)
45.34
>>>type(x)
<class 'float'>
```

Data types -1

boolean

boolean is a value that represents one of two states, True or False, being equal to 1 or 0 respectively . Note also that any number other than 0 can represent the True state .

```
>>>x = True
>>>print(x)
True
>>>type(x)
<class 'bool'>
>>>bool(1)
True
```

```
>>>x = False
>>>print(x)
False
>>>type(x)
<class 'bool'>
>>>bool(0.0)
False
```

Data types -1

None

None is a reserved keyword in python used to define a no value . None is not the same as 0, False, or an empty string. None is a data type of its own (Nonetype) and only None can be None .

```
>>>x = None

>>>type(x)

<class 'Nonetype'>

>>>bool(x)

False

>>>None == False

False
```

```
>>>None == None

True

>>>None != None

False
```

Arithmetic operations

Addition

+

Grow Through

Arithmetic operations

Subtraction

Grow Through

Arithmetic operations

Multiplication

*

strings can also be multiplied by an integer n resulting a repeated string n times

Arithmetic operations

Division

/

For integer division we use double slash instead of a single one

Arithmetic operations

Modulo

%

Modulus also works with floating point numbers in python

Grow Through

Arithmetic operations

Modulo

Q: Given the periodic sin(input) that only works within the period range [-pi, pi], modify the input so the given sin function can calculate beyond this range .

input = (input + pi) % 2*pi - pi

Arithmetic operations

Power

**

Grow Through

Data types -2

Collections

- List is a collection which is ordered and changeable. Allows duplicate members.
- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- Set is a collection which is unordered and unindexed. No duplicate members.
- Dictionary is a collection which is unordered, changeable and indexed. No duplicate members.

Data types -2

list

An ordered collection .

```
>>>x = [1,2,3,'and a string']
>>>print(x)
[1, 2, 3, 'and a string']
>>>x[1]
2
>>>x[1] = 2.2
>>>print(x)
[1, 2.2, 3, 'and a string']
```

Data types -2

tuple

An ordered & unchangeable collection .

```
>>>x = (1,2,3,'and a string')
>>>print(x)
(1, 2, 3, 'and a string')
>>>x[1]
2
>>>x[1] = 2.2
TypeError: 'tuple' object does not support item assignment
```

Data types -2

set

An unordered & unindexed collection .

```
>>>x = {1,2,3,'and a string'}
>>>print(x)
{1, 2, 3, 'and a string'}
>>>x[1]
TypeError: 'set' object does not support indexing
```

Data types -2

dict

An unordered & indexed collection .

```
>>>x = {1:4,2:'two',3:3,'and a string':'mapping to another string'}
>>>print(x)
{1:4, 2:'two', 3:3, 'and a string':'mapping to another string'}
>>>x[1]
4
>>>x['and a string']
'mapping to another string'
```

Logic and execution flow

Equality

The equality comparison returns True or False based on the two sides of the comparison .

```
>>>x = 1

>>>x == 1

True

>>>x == 1.0

True

>>>x == '1'

False
```

Logic and execution flow

Less or greater than

```
>>>x = 1
>>>x > 1
False
>>>x >= 1.0
True
>>>x < '1'
TypeError: '<' not supported between instances of 'int' and 'str'
```

Logic and execution flow

Inequality

```
>>>x = 1
>>>x != 1
False
>>>x != 2
True
>>>x != '1'
True
```

Grow Through

Logic and execution flow

is, in & not

is keyword is used to to check whether two variables referring to the same object or not, while in is used to check whether an object is present inside another object or not .

is and in keywords can be suffixed and prefixed respectively with not to alter their behavior .

not keyword is equivalent to the exclamation mark (!) in other languages .

```
>>>not True

False
```

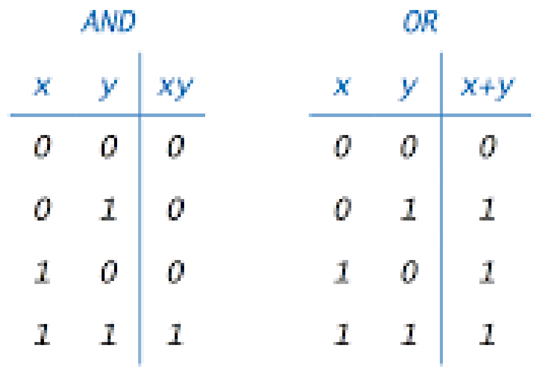
Logic and execution flow

or & and

or operation returns True if at least one of its operands is True .

and operation returns True if and only if all of its operands are True .

```
>>>True or False
True
>>>True and False
False
>>>True and True
True
```



AND

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

OR

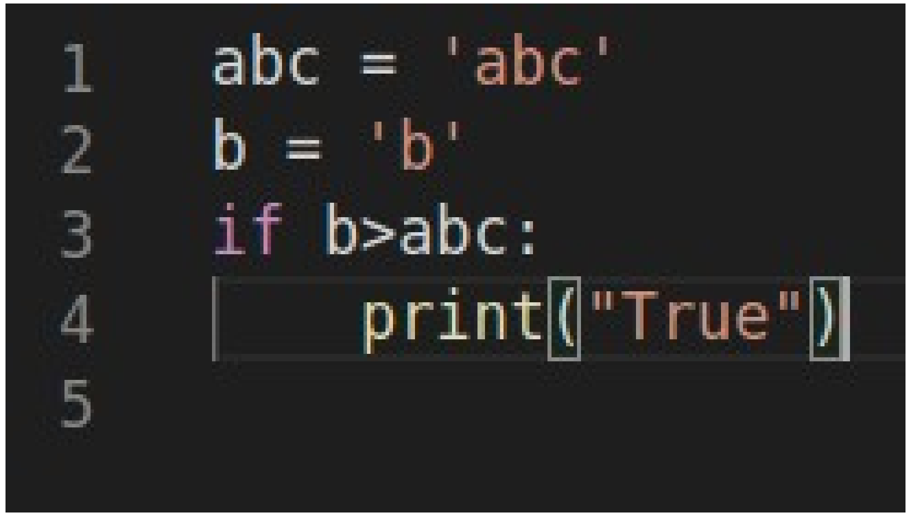
x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

and & or truth table

Logic and execution flow

if

An if statement executes if and only if its conditional expression returns True .



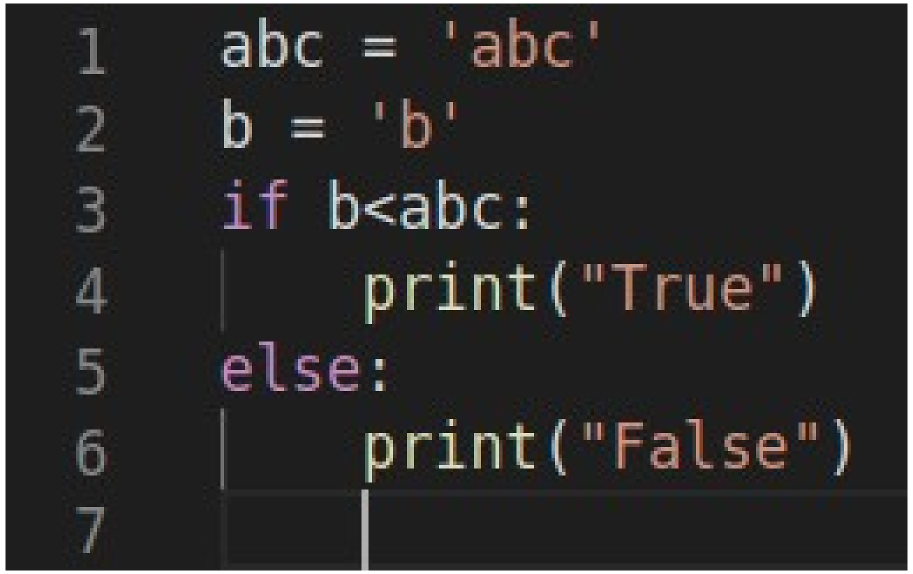
```
abc = 'abc'
b = 'b'
if b>abc:
    print("True")
```

Outputs True because strings are compared by their lexicographical order .

Logic and execution flow

else

Else keyword may follow an if statement to determine what happens in case the if statement wasn’t executed .



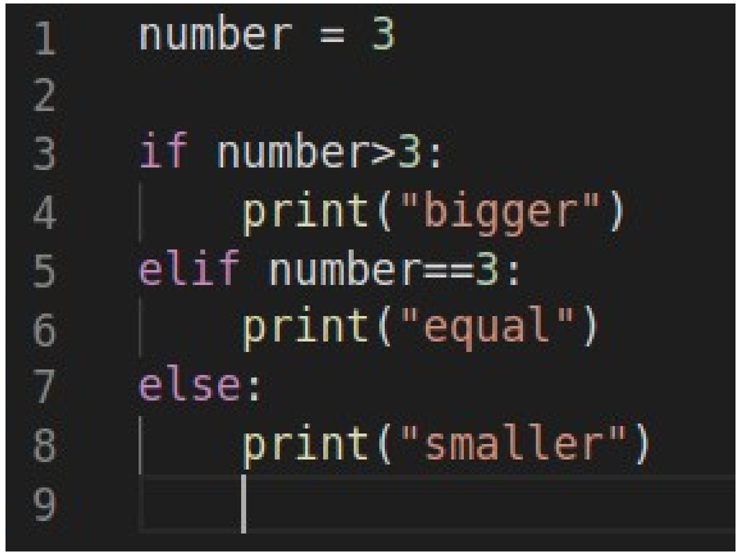
```
abc = 'abc'
b = 'b'
if b<abc:
    print("True")
else:
    print("False")

```

Logic and execution flow

elif

Because the indentation matters in python, each single else-if statement would need an extra indentation tab, would result in a bad looking code .

An elif keyword doesn’t need extra indentation so would make a much cleaner code .

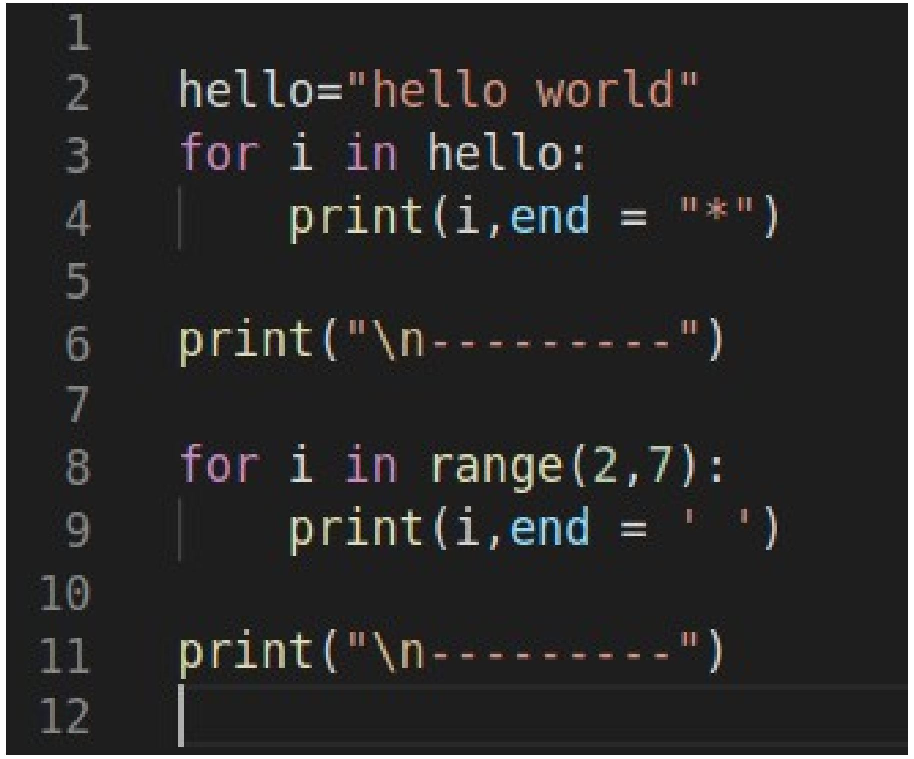


```
number = 3

if number>3:
    print("bigger")
elif number==3:
    print("equal")
else:
    print("smaller")
```

Outputs “equal”

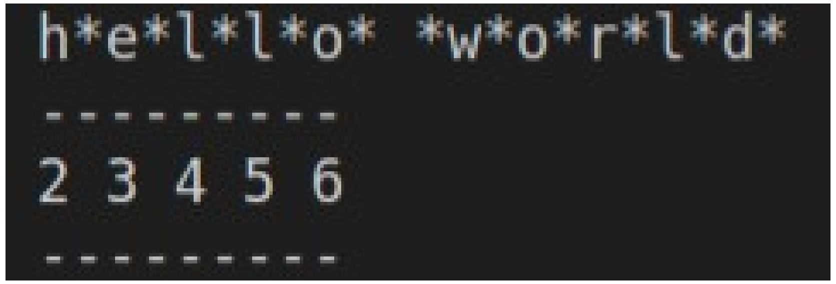
Logic and execution flow

for



```

hello="hello world"
for i in hello:
    print(i,end = "*")

print("\n---------")

for i in range(2,7):
    print(i,end = ' ')

print("\n---------")

```

Outputs

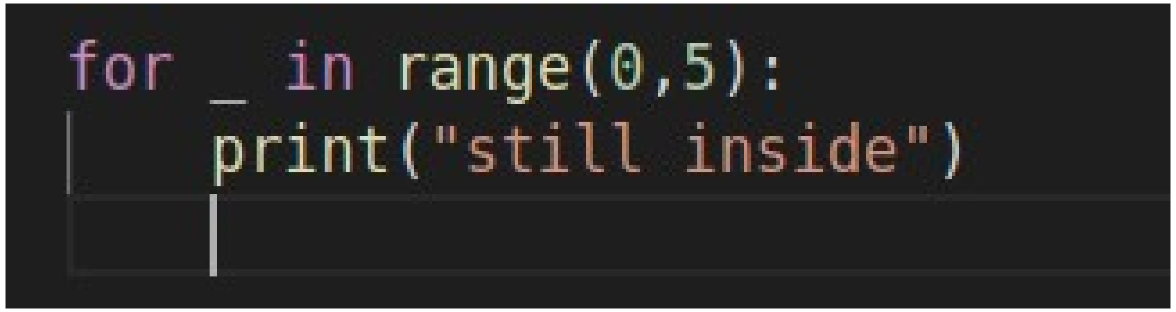


```
h*e*l*l*o* *w*o*r*l*d*
---------
2 3 4 5 6
---------
```

Logic and execution flow

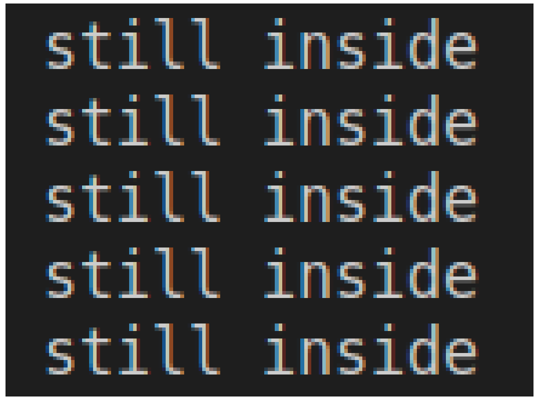
_

When we have no interest in the value of the iterator in each iteration, we can use _ which just ignores (doesn't store) the iteration's value .



```python
for _ in range(0,5):
    print("still inside")
```

Outputs



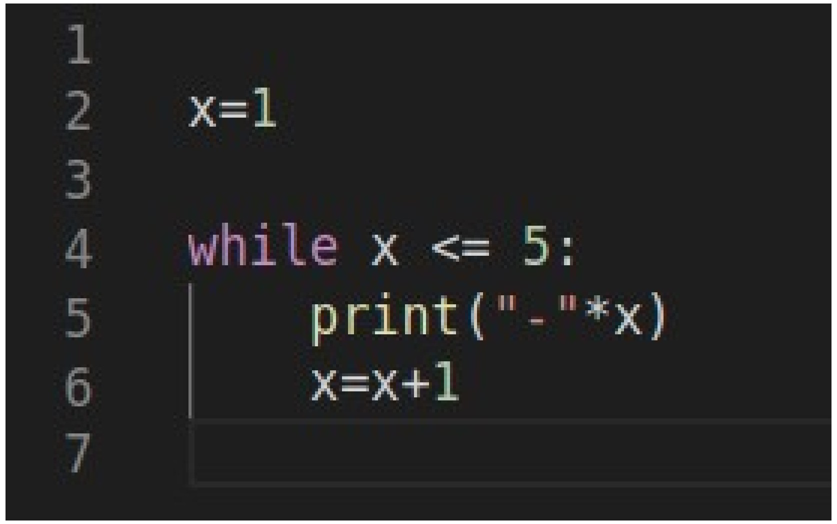
```
still inside
still inside
still inside
still inside
still inside
```

Grow Through

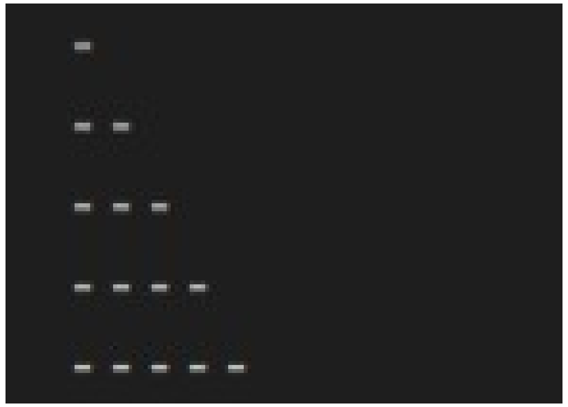
Logic and execution flow

while

A while loop keeps executing its scope until its condition is False, the loop stops at this point .



```

x=1

while x <= 5:
    print("-"*x)
    x=x+1

```



```
-
--
---
----
-----
```

Grow Through

Logic and execution flow

.=

A simpler way to write x=x+1 is x+=1, the same applies for all other arithmetic operations .

E.g.

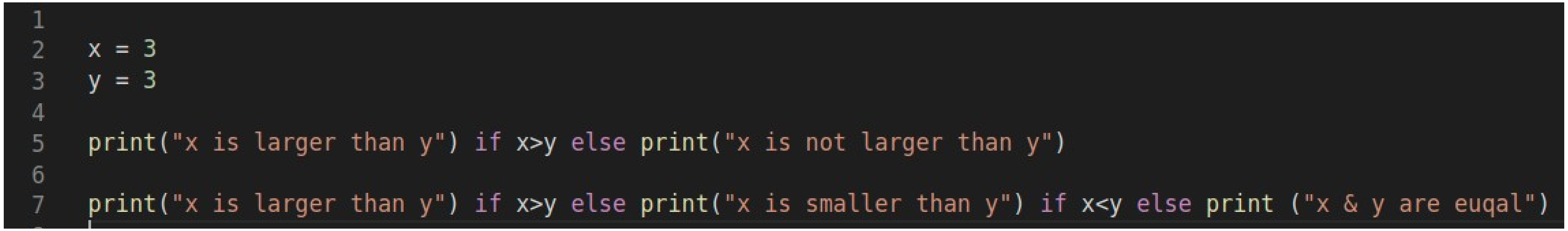
```
x=x**2      is the same as     x**=2
                 also
x=x//7      is the same as     x//=7
```

And so on …

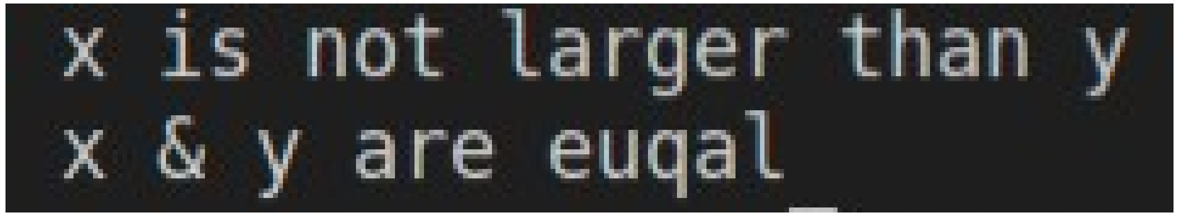
Logic and execution flow

Ternary operator

Python doesn't have the conventional ternary operator "condition ? true_exe:false_exe " but has a more descriptive statement : "true_exe if condition else false_exe" .



```python

x = 3
y = 3

print("x is larger than y") if x>y else print("x is not larger than y")

print("x is larger than y") if x>y else print("x is smaller than y") if x<y else print ("x & y are euqal")
```

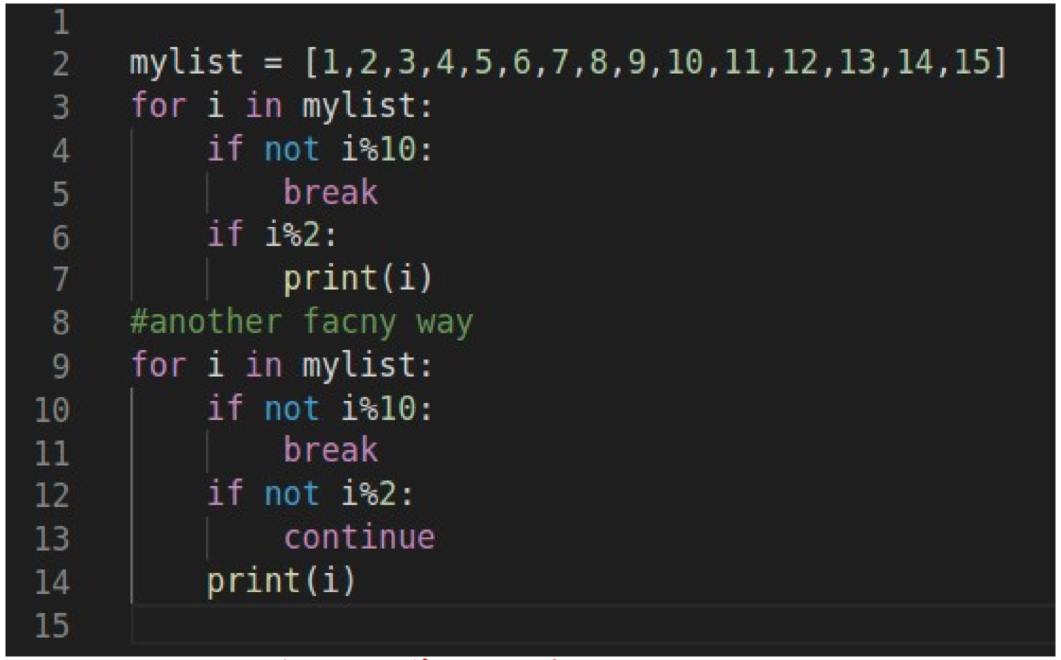


```
x is not larger than y
x & y are euqal
```

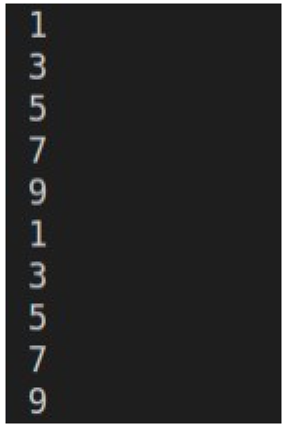
Logic and execution flow

break & continue

break keyword exits the enclosing running loop, and the continue statement causes the interpreter to stop executing the current iteration and start from the beginning of the loop .



```python

mylist = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
for i in mylist:
    if not i%10:
        break
    if i%2:
        print(i)
#another facny way
for i in mylist:
    if not i%10:
        break
    if not i%2:
        continue
    print(i)

```



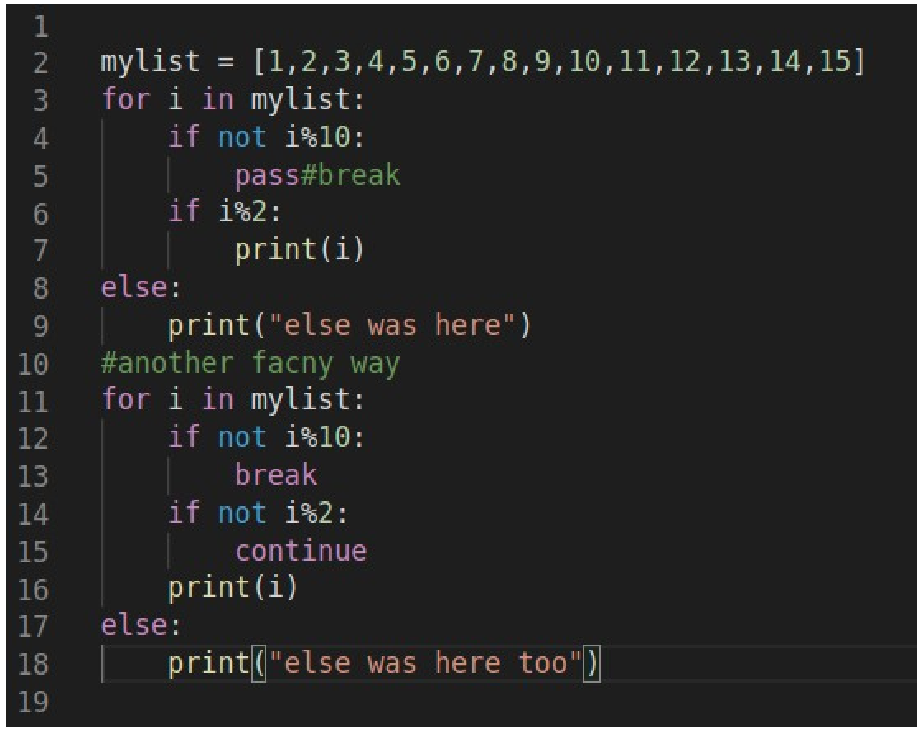
```
1
3
5
7
9
1
3
5
7
9
```

Grow Through

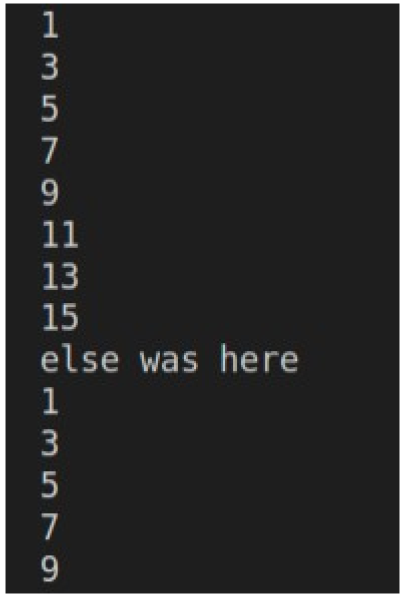
Logic and execution flow

else after loops

An else statement after a loop would execute if the condition of the loop ever evaluates to False .



```python

mylist = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
for i in mylist:
    if not i%10:
        pass#break
    if i%2:
        print(i)
else:
    print("else was here")
#another facny way
for i in mylist:
    if not i%10:
        break
    if not i%2:
        continue
    print(i)
else:
    print("else was here too")

```



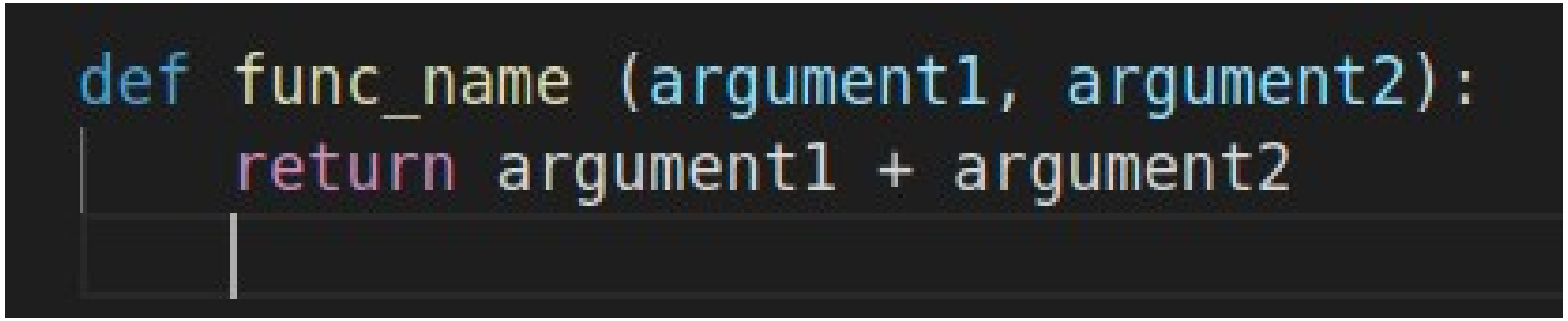
```
1
3
5
7
9
11
13
15
else was here
1
3
5
7
9
```

Grow Through

Functions

Definition

A function is defined with def keyword followed by the name of the function followed by a couple of parentheses that holds the function's arguments .

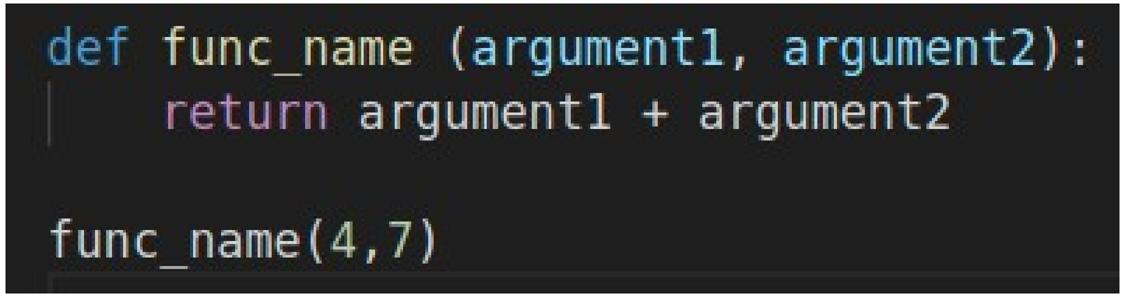


```
def func_name (argument1, argument2):
    return argument1 + argument2
```

Functions

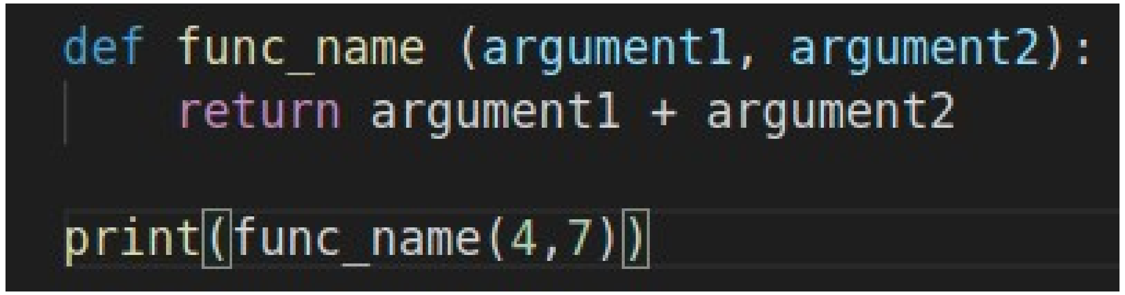
Invocation

We write the name of the function followed by a couple of parentheses holding the passed arguments or empty parentheses in case the function can take no arguments .



```
def func_name (argument1, argument2):
    return argument1 + argument2

func_name(4,7)
```

But this code won't output the return value because we didn't print It, when we run python file any return value for each line of code is not outputted like in the interpreter .



```
def func_name (argument1, argument2):
    return argument1 + argument2

print(func_name(4,7))
```

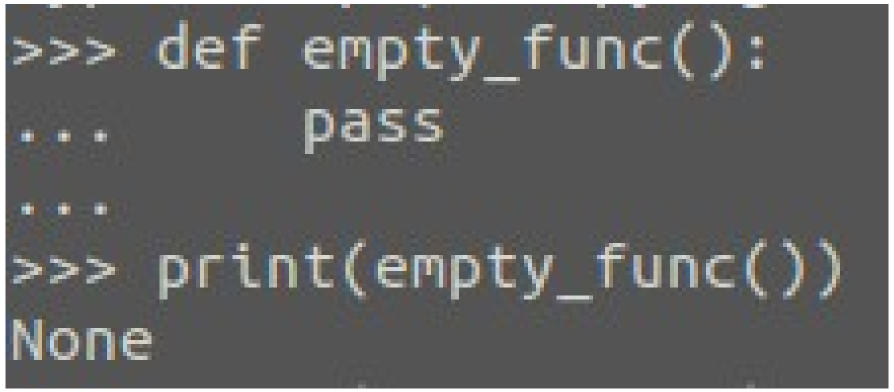
Outputs 11

Grow Through

Functions

Empty function

We can’t write a function in python and just leave it empty . We use pass keyword in this case :



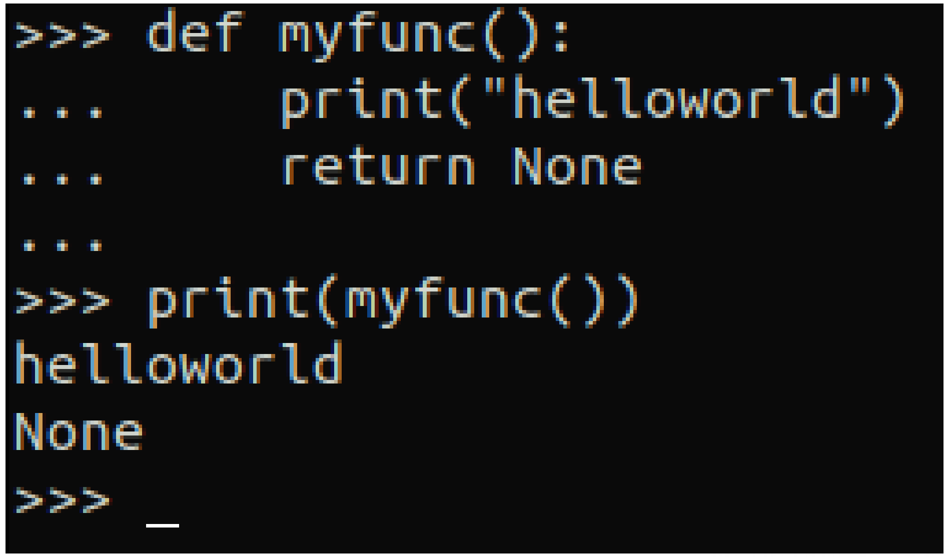
```
>>> def empty_func():
...     pass
...
>>> print(empty_func())
None
```

Because such a function doesn’t return anything in this case, when we invoke it inside print() we get None, which means no value.

Functions

return

return keyword is used to return a value to the caller of the function .We can use it multiple times in one function based on some sort of conditions and we can never use it at all which in this case the function would finish executing and return None, we can also explicitly return None .



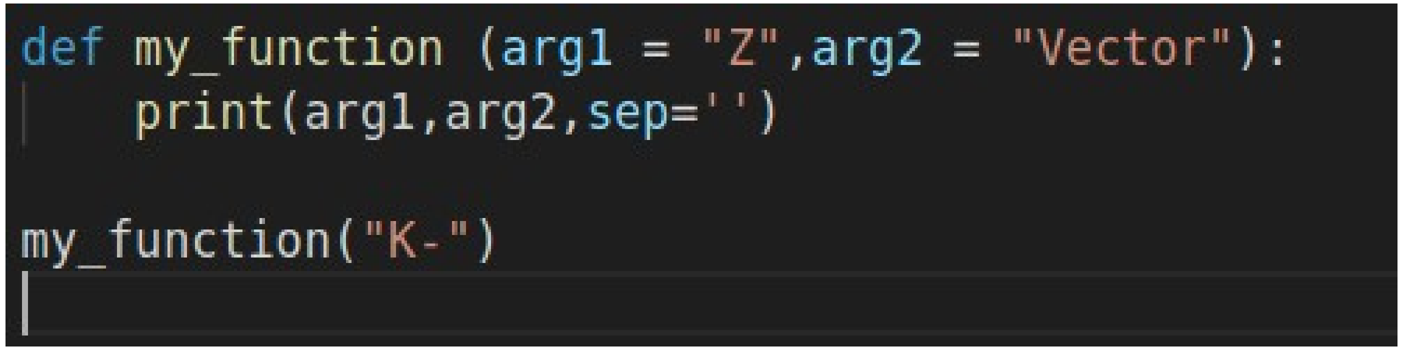
```
>>> def myfunc():
...     print("helloworld")
...     return None
...
>>> print(myfunc())
helloworld
None
>>> _
```

None doesn’t output in the interpreter unless it’s printed .

Functions

Default arguments

An argument can have a default value to be used in case no value was passed for its parameter to the function .



```
def my_function (arg1 = "Z",arg2 = "Vector"):
    print(arg1,arg2,sep='')

my_function("K-")
```

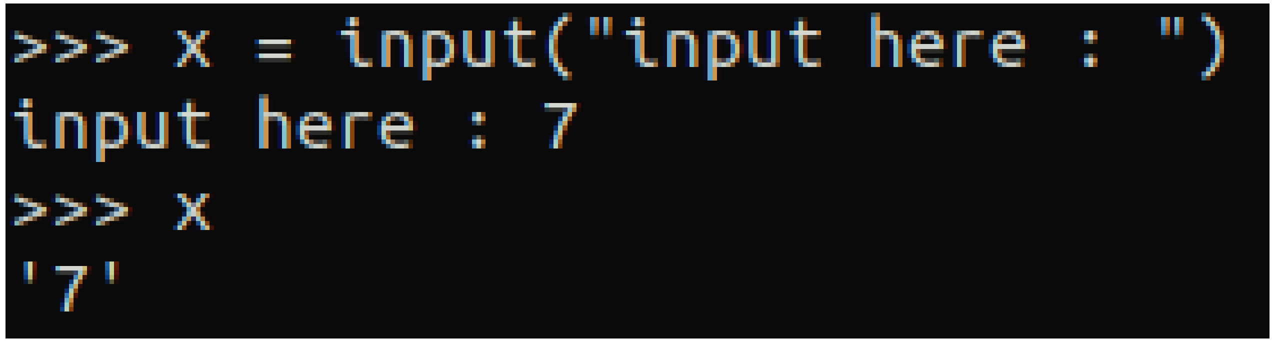
The code above outputs “K-Vector”, the first parameter has a passed argument but the second parameter doesn’t have an argument so it uses its default argument .

Functions

print() & input()

print() is used to output the value it gets to the console, it behaves differently with different data types.

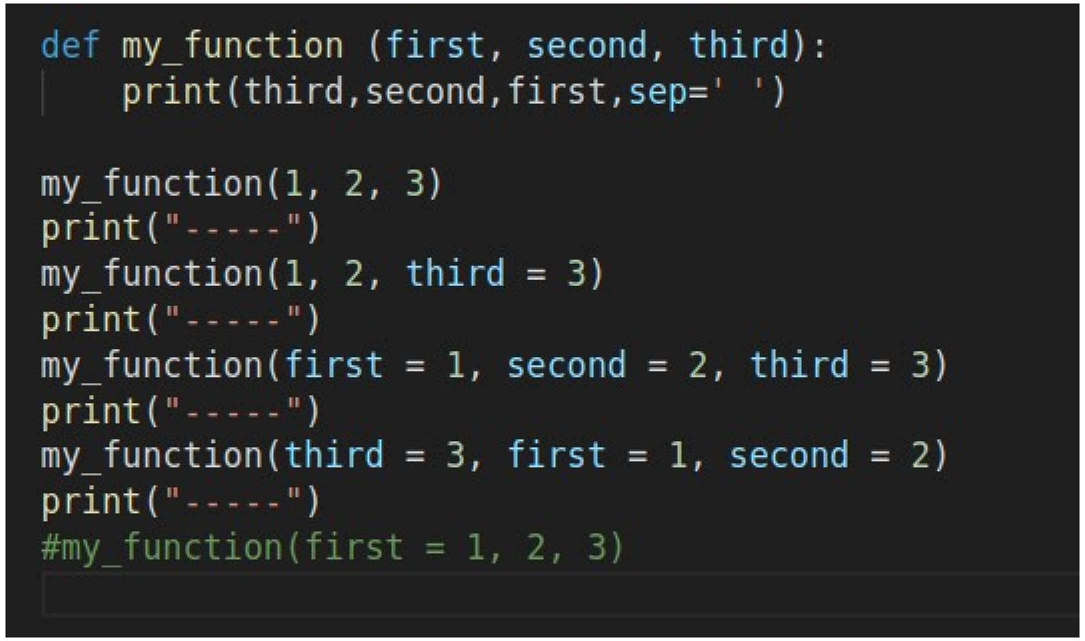
input() in the other hand is used to retrieve an input from the console and it only gets the input as a string, so sometimes we need to convert it to a different data type before working with it, input also takes an argument and outputs it before asking for input .



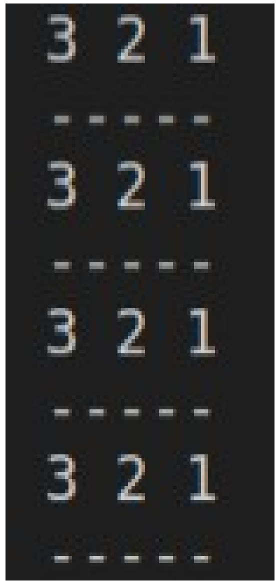
```
>>> x = input("input here : ")
input here : 7
>>> x
'7'
```

Functions

Positional and named arguments



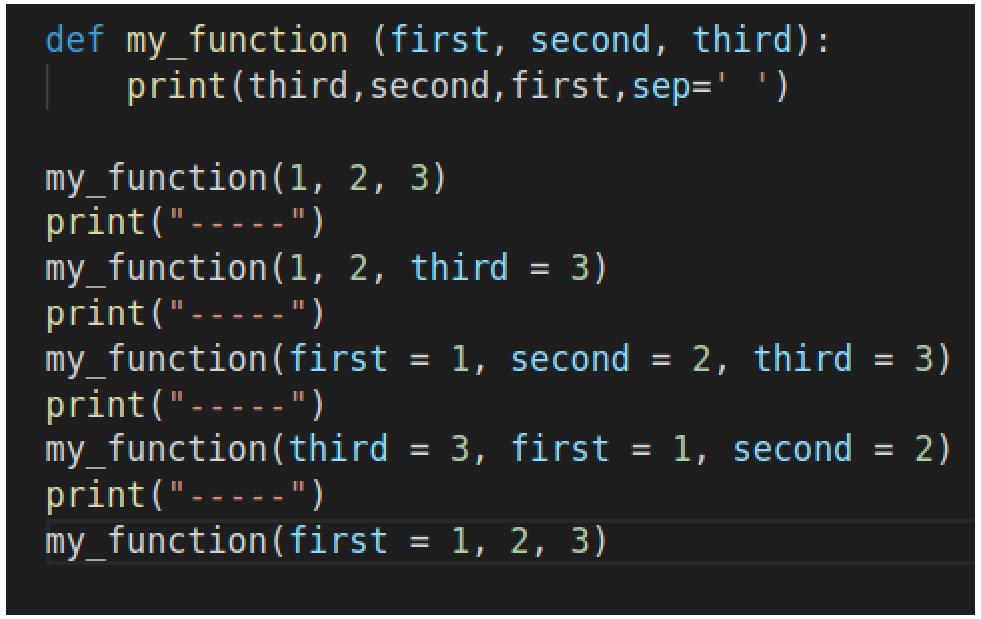
```
def my_function (first, second, third):
    print(third,second,first,sep=' ')

my_function(1, 2, 3)
print("-----")
my_function(1, 2, third = 3)
print("-----")
my_function(first = 1, second = 2, third = 3)
print("-----")
my_function(third = 3, first = 1, second = 2)
print("-----")
#my_function(first = 1, 2, 3)
```



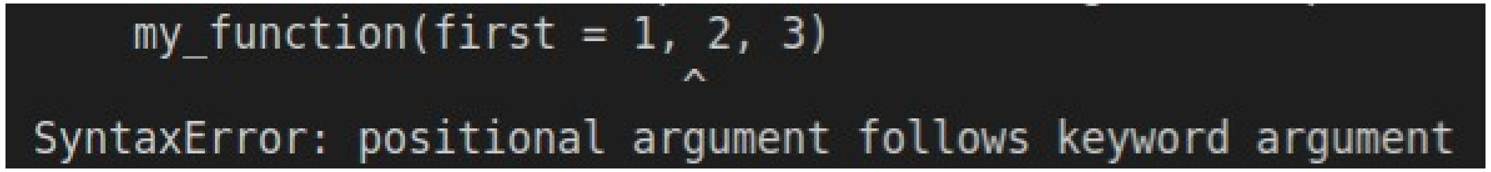
```
3 2 1
-----
3 2 1
-----
3 2 1
-----
3 2 1
-----
```

Functions

Positional and named arguments



```
def my_function (first, second, third):
    print(third,second,first,sep=' ')

my_function(1, 2, 3)
print("-----")
my_function(1, 2, third = 3)
print("-----")
my_function(first = 1, second = 2, third = 3)
print("-----")
my_function(third = 3, first = 1, second = 2)
print("-----")
my_function(first = 1, 2, 3)
```



```
    my_function(first = 1, 2, 3)
                           ^
SyntaxError: positional argument follows keyword argument
```

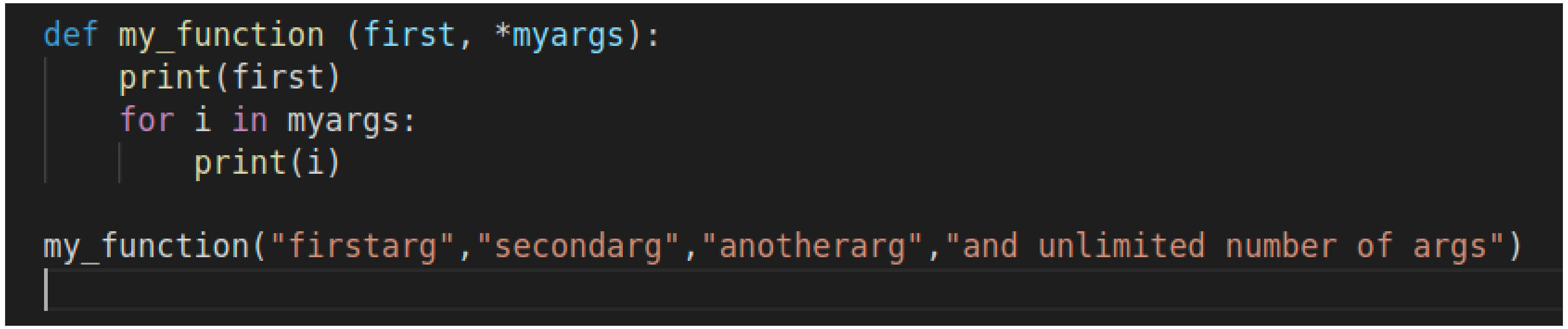
The code won't compile with syntax errors, so we can't even see the output of the first few lines although the error is in the last line .

Grow Through

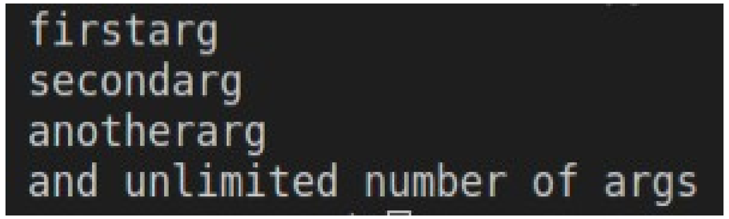
Functions

*args

*args is used to collect an unlimited number or positional arguments and stores them inside a tuple called args .



```
def my_function (first, *myargs):
    print(first)
    for i in myargs:
        print(i)

my_function("firstarg","secondarg","anotherarg","and unlimited number of args")
```

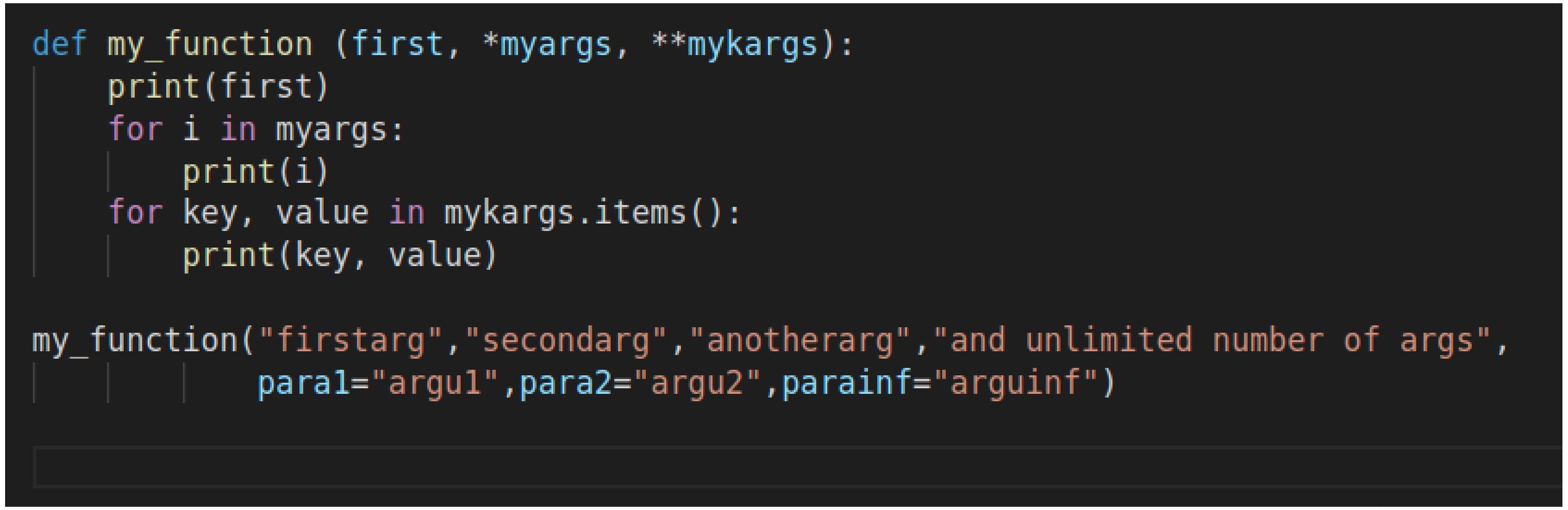


```
firstarg
secondarg
anotherarg
and unlimited number of args
```

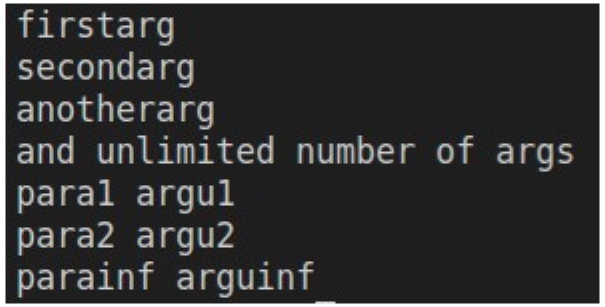
Functions

**kargs

*kargs is used to collect an unlimited number or named arguments and stores them inside a dictionary called kargs with the key being the parameter and the value being the argument .



```
def my_function (first, *myargs, **mykargs):
    print(first)
    for i in myargs:
        print(i)
    for key, value in mykargs.items():
        print(key, value)

my_function("firstarg","secondarg","anotherarg","and unlimited number of args",
            para1="argu1",para2="argu2",parainf="arguinf")
```

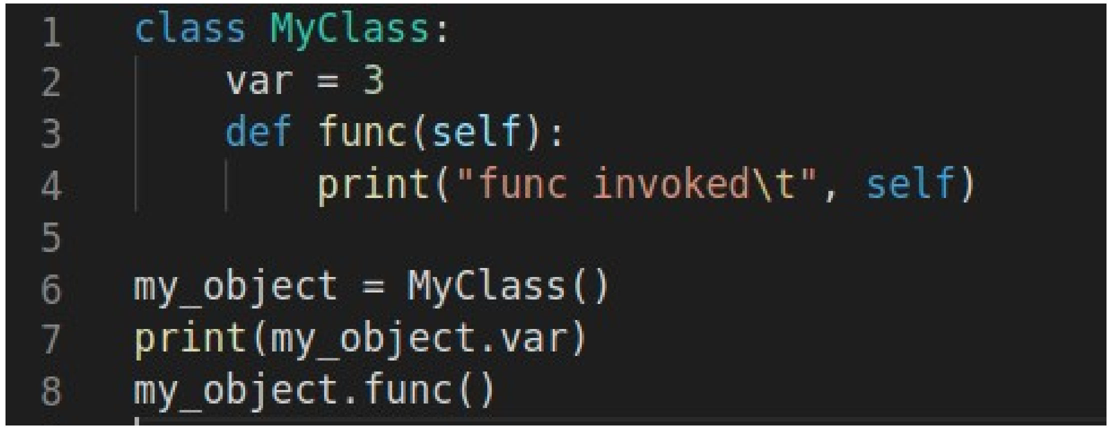


```
firstarg
secondarg
anotherarg
and unlimited number of args
para1 argu1
para2 argu2
parainf arguinf
```

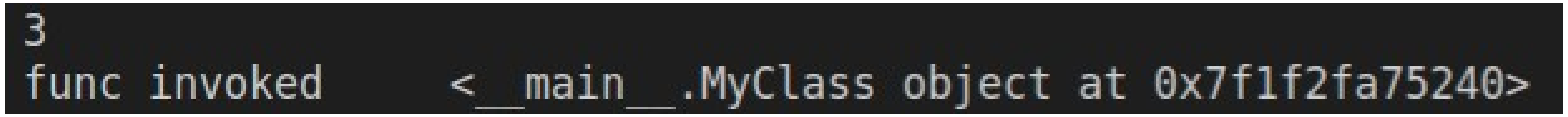
Classes

definition

Classes are defined with class keyword .



```python
class MyClass:
    var = 3
    def func(self):
        print("func invoked\t", self)

my_object = MyClass()
print(my_object.var)
my_object.func()
```

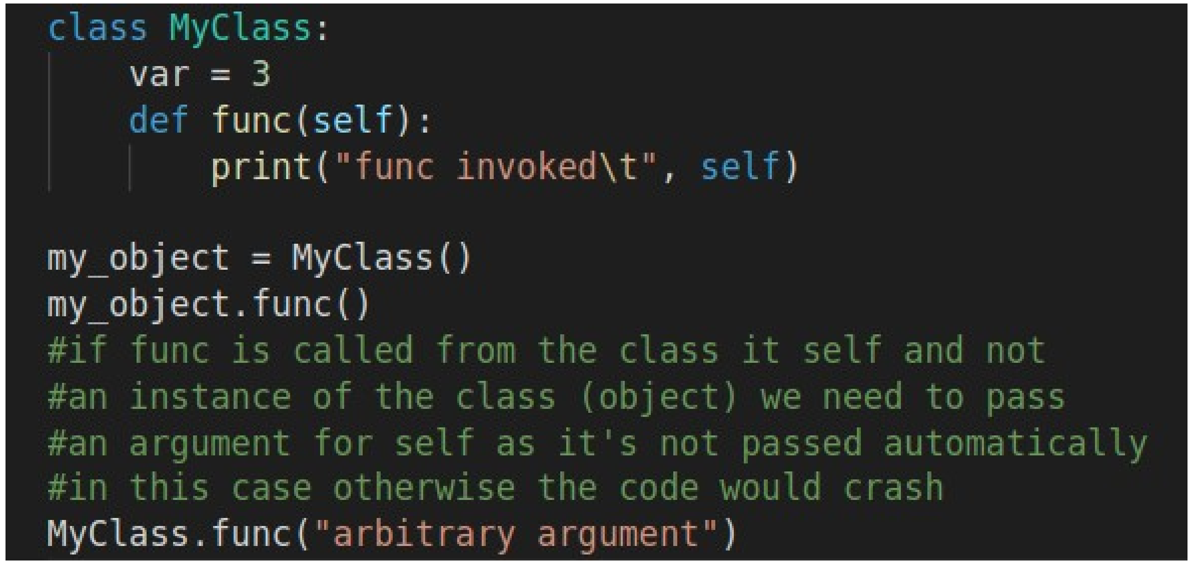


```
3
func invoked     <__main__.MyClass object at 0x7f1f2fa75240>
```

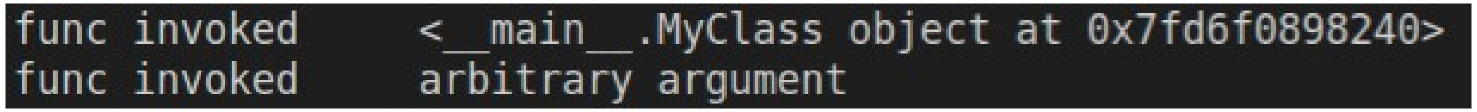
Classes

self

self is a keyword referring to the object instantiated from that class. Any function in the class would be passed self as the first argument if it was called by an object of the class.



```python
class MyClass:
    var = 3
    def func(self):
        print("func invoked\t", self)

my_object = MyClass()
my_object.func()
#if func is called from the class it self and not
#an instance of the class (object) we need to pass
#an argument for self as it's not passed automatically
#in this case otherwise the code would crash
MyClass.func("arbitrary argument")
```



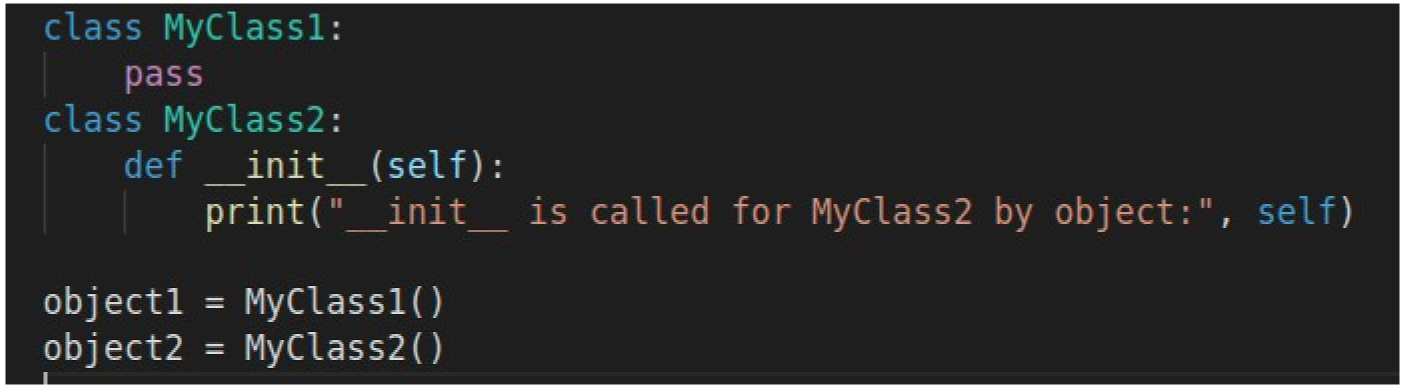
```
func invoked     <__main__.MyClass object at 0x7fd6f0898240>
func invoked     arbitrary argument
```

Grow Through

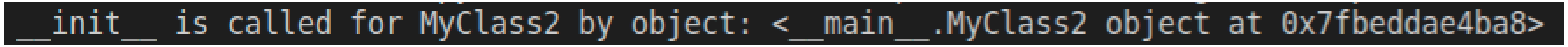
Classes

__init__()

__init__(self, *args, *kargs) is the function getting called once an instance of the class is made (constructed) .



```
class MyClass1:
    pass
class MyClass2:
    def __init__(self):
        print("__init__ is called for MyClass2 by object:", self)

object1 = MyClass1()
object2 = MyClass2()
```

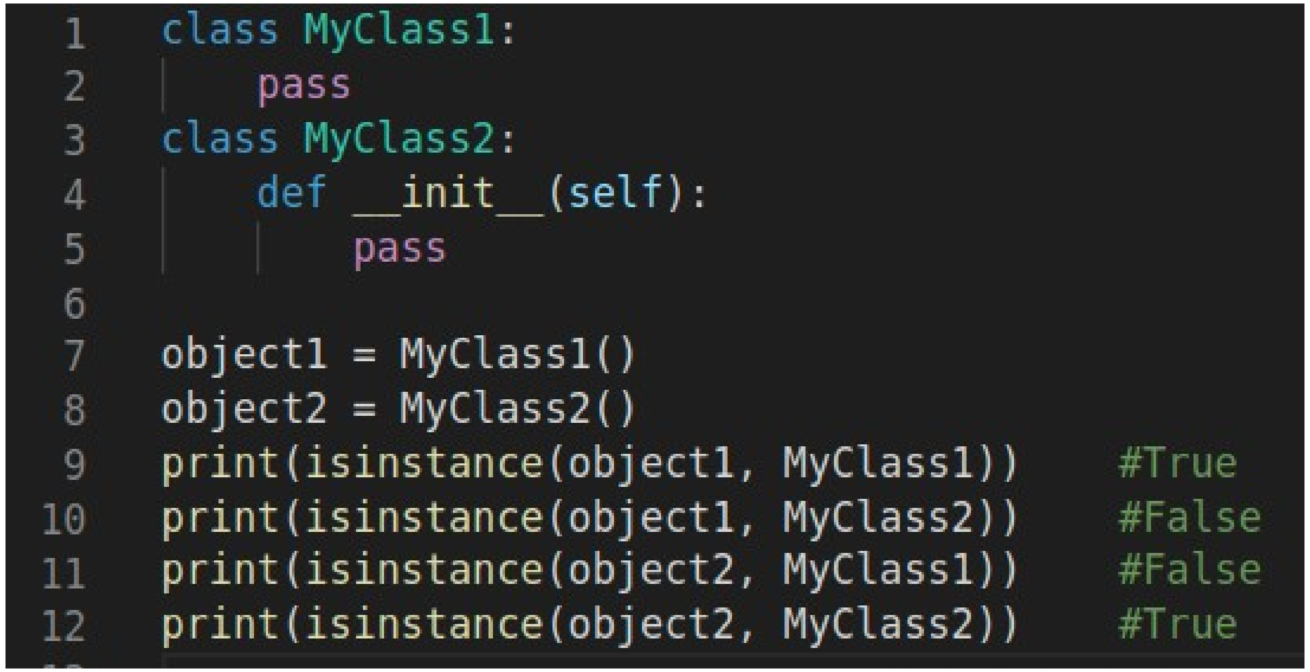


```
__init__ is called for MyClass2 by object: <__main__.MyClass2 object at 0x7fbeddae4ba8>
```

Classes

isinstance()

isinstance(object, class) function is used to check whether an object is of a particular class or not .



```
class MyClass1:
    pass
class MyClass2:
    def __init__(self):
        pass

object1 = MyClass1()
object2 = MyClass2()
print(isinstance(object1, MyClass1))    #True
print(isinstance(object1, MyClass2))    #False
print(isinstance(object2, MyClass1))    #False
print(isinstance(object2, MyClass2))    #True
```

Thank You