

Software Engineering – Architectures, Designs, and Code

L01 on Coding

Giancarlo Succi

Dipartimento di Informatica – Scienza e Ingegneria

Università di Bologna

`g.succi@unibo.it`

Location of the material:

`https://github.com/GiancarloSucci/UniBo.SE.A2024`



Structure of the Part on Coding

- Part 0:
 - Revision of the strategy pattern in Java (and, a very little bit, on C++)
- Part 1:
 - Introduction on Python as a self-proclaimed multiparadigm language
- Part 2:
 - Patterns in Python with an eye on ChatGPT and around



Caveat on Python (1/2)

- Here we will reflect on the concepts we saw on architecture and design looking at the code
- We will focus on a concrete example, which could be used also in the overall course
- We will see many examples of code
- Also very detailed, step-by-step examples
- Our focus will be on the software engineering side, though
- That is in understanding why and how people achieved what they achieved
 - not on hacking solutions



Caveat on Python (2/2)

- The most liked programming languages by the instructor
 - 1. C++
 - 2. C
 - 3. Java
 - 4. Haskell
 - 5. SmallTalk
 - 6. ...
 - ...
 - n-1. ...
 - n. Python
- We use Python because it is becoming a standard
- But we start with a mention of the ancestor



Python for our purposes

- About the language
- Structure of the execution of Python
- Virtual Environment



About the language

- Origin and principles of the language
- Fundamental syntax
- Structure of execution
- String formatting
- Object orientation
- Polymorphism and late binding
- Functions as objects
- Decorators
- Static members
- Structure of the virtual machine



Origin of the language

- The language was conceived by Guido van Rossum at CWI in the '80s, got its first implementation in the early '90, and then was modified and upgraded multiple times
- The latest stable version of Python (at the time of writing these slides) is 3.11.5, released on 5th October 2022
- The language is released in a “kind of” open source license, the Python Software Foundation License but there are debates about it
- Apparently, Python comes from ABC and SETL (see https://en.wikipedia.org/wiki/History_of_Python)
- The instructor sees a strong resemblance of SmallTalk (see the references below)
- **References:**
 - Python vs. Smalltalk: from StackOverflow and from Medium.
 - About Python in wikipedia: [https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language)), https://en.wikipedia.org/wiki/History_of_Python and related pages



Principles of the language

- Multiparadigm (scripting, imperative, object oriented, and functional)
- Dynamically typed
- Garbage collected
- “Batteries included”
- Based on a virtual machine (“kind of” interpreted)
- **References:**
 - About Python in wikipedia: [https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))



The Zen of Python (1/2)

- 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.

- **References:**

- About the Zen of Python in wikipedia: https://en.wikipedia.org/wiki/Zen_of_Python



The Zen of Python (2/2)

- By Tim Peters
 - 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!

- **References:**

- About the Zen of Python in wikipedia: https://en.wikipedia.org/wiki/Zen_of_Python



Multiparadigm

- We will now explore Python in its 4 paradigms
 - Scripting
 - Imperative
 - Functional
 - Object Oriented
- With such an approach we will cycle over the syntax and the semantics of the language to get a comprehensive view of it and revising the fundamental principles of software engineering



Python as a scripting language



Scripting (1/3)

- Install the Python interpreter: multiple options including
 - Mac: see the guidelines in **Pip Upgrade – And How to Update Pip and Python**
 - Windows: like here: **pyenv for Windows**
- Open the Python interpreter ...
- ... and try :)
- Analysing the scripting perspective of Python exposes us to the fundamental control structures
 - variables, if, blocks via indentation, while, for, collections, iterations



Scripting (2/3)

```
% python3.11
Python 3.11.2 (v3.11.2:878ead1ac1, Feb 7 2023, 10:02:41) [Clang 13.0.0 (clang
-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>> 4+3
7
>>> x = 7+1
>>> x**2
64
>>>
```



Scripting (3/3)

```
% python3.11
Python 3.11.2 (v3.11.2:878ead1ac1, Feb 7 2023, 10:02:41) [Clang 13.0.0 (clang
-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
>>> ^D
```

Source: <https://docs.python.org/3/tutorial/interpreter.html>



Indentation

- If I forget the indentation ...

```
% python3.11
Python 3.11.2 (v3.11.2:878ead1ac1, Feb 7 2023, 10:02:41) [Clang 13.0.0 (clang
-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...   print("Be careful not to fall off!")
      File "<stdin>", line 2
        print("Be careful not to fall off!")
      ~
IndentationError: expected an indented block after 'if' statement on line 1
>>> ^D
```

Source: <https://docs.python.org/3/tutorial/interpreter.html>



Dynamic Typing

- Typing is dynamic and enforced ...

```
% python3.11
Python 3.11.2 (v3.11.2:878ead1ac1, Feb 7 2023, 10:02:41) [Clang 13.0.0 (clang
-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> x = "a string"
>>> x = 1
>>> print(x)
1
>>> y = 1 + "a string"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Source: <https://docs.python.org/3/tutorial/introduction.html>



Math

- Mathematics follows mostly the usual approaches ...

```
% python3.11
>>> 3+4
7
>>> 6-3.4
2.6
>>> 4*7.0
28.0
>>> 7/3
2.3333333333333335
>>> 7//3
2
>>> 7%3
1
>>> 7**3
343
>>> 3 + _
346
>>>
```



Strings (1/2)

- Strings are a bit more peculiar ...

```
>>> x='strings can be single quoted '  
>>> y="or double quoted "  
>>> x + y + "and joined with + "  
'strings can be single quoted or double quoted and joined with + '  
>>>  
>>> z=''' strings can span  
... multiple lines using triple  
...  
... quotes'''  
>>> z  
' strings can span\nmultiple lines using triple\n\nquotes'  
>>> print(z)  
strings can span  
multiple lines using triple  
  
quotes  
>>>
```

Source: <https://docs.python.org/3/tutorial/introduction.html>



Strings (2/2)

- Strings are a bit more peculiar ...

```
>>> w='And I can add \n special chars like in C, which are printed using print'
>>> w
'And I can add \n special chars like in C, which are printed using print'
>>> print(w)
And I can add
  special chars like in C, which are printed using print
>>> c='concatenation ' 'is done just sequencing strings '
>>> c
'concatenation is done just sequencing strings '
>>> c ' but not using variables'
File "<stdin>", line 1
  c ' but not using variables'
  ~~~~~
SyntaxError: invalid syntax
>>> 2*' and the n* operator repeats the string n times'
' and the n* operator repeats the string n times and the n* operator repeats the
  string n times'
```

Source: <https://docs.python.org/3/tutorial/introduction.html>



Indexing Strings (1/2)

- Strings can be indexed ...

```
>>> e='My Example of Python'
>>> e[0]
'M'
>>> e[19]
'n'
>>> e[20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> e[-1]
'n'
>>> e[-19]
'y'
>>> e[-20]
'M'
>>> e[3:5]
'Ex'
>>> e[-4:-1]
'tho'
```



Indexing Strings (2/2)

- Ranges are start-included, end-excluded ...
- Slicing can include the step as third parameter

```
>>> e[0:3]
'My '
>>> e[11:]
'of Python'
>>> e[:11]+e[11:]
'My Example of Python'
>>> e[2:10:2]
' xml'
>>> e[::2]
'M xml fPto'
>>> e[::-1]
'nohtyP fo elpmaxE yM'
```

Source: <https://docs.python.org/3/tutorial/introduction.html>



More on Strings

- Strings are immutable

```
>>> e[0]='T'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> w = 'T' + e[1:]
>>> w
'Ty Example of Python'
>>>
```

- Length of a string

```
>>> len(w)
20
>>> len("123")
3
```

Source: <https://docs.python.org/3/tutorial/introduction.html>



Collections

- Python does not have arrays
- Python has 4 basic collection types, partly with syntax similar to that of strings:
 - Lists: ordered, mutable collections with duplicates
 - Tuples: ordered, immutable collection with duplicates
 - Sets: unordered collection, without duplications, whose members are immutable
 - Dictionaries: ordered, mutable collections without duplicates

Source: https://www.w3schools.com/python/python_tuples.asp



Lists (1/8)

- List are ordered collections of elements, with a syntax resembling that of strings

```
>>> odd = [1, 3, 5, 7, 9]
>>> odd
[1, 3, 5, 7, 9]
>>> odd[0]
1
>>> odd[4]
9
>>> odd[-1]
9
>>> odd[-5]
1
>>> odd[1:3]
[3, 5]
>>> odd[:2]+odd[2:]
[1, 3, 5, 7, 9]
>>> len(odd)
5
```



Lists (2/8)

- List are mutable, append-able, slice-able

```
>>> odd[2]=20
>>> odd
[1, 3, 20, 7, 9]
>>> odd.append(11)
>>> odd
[1, 3, 20, 7, 9, 11]
>>> odd[3:5]=[100,200,300]
>>> odd
[1, 3, 20, 100, 200, 300, 11]
>>> odd[1:3] = []
>>> odd
[1, 100, 200, 300, 11]
>>> odd[:] = []
>>> odd
[]
```

Source: <https://docs.python.org/3/tutorial/introduction.html>



Lists (3/8)

- List are nestable and heterogeneous

```
>>> a = [1,2,3]
>>> b = [10,20]
>>> c = [90, 800, -34]
>>> n = [a, b, c]
>>> n
[[1, 2, 3], [10, 20], [90, 800, -34]]
>>> n[2][1]
800
>>> z = [1, "xxx", 3]
>>> z
[1, 'xxx', 3]
>>> z[1]
'xxx'
```

Source: <https://docs.python.org/3/tutorial/introduction.html>



Lists (4/8)

- Copy of references

```
>>> x = [1, 2, 3, 4]
>>> y = x
>>> y[2]=200
>>> x
[1, 2, 200, 4]
```

Source: <https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/>



Lists (5/8)

● Shallow copies

```
>>> w = x.copy()
>>> w[1]=350
>>> w
[1, 350, 200, 4]
>>> x
[1, 2, 200, 4]
```

Source: <https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/>



Lists (6/8)

● Shallow copies (more)

```
>>> z = [11,22,33]
>>> k = [121, 132, 143, 154]
>>> j = [z,k]
>>> j
[[11, 22, 33], [121, 132, 143, 154]]
>>> i = j.copy()
>>> k[1] = 222
>>> i
[[11, 22, 33], [121, 222, 143, 154]]
>>> i[0] = [5, 8, 13]
>>> k[1] = 222
>>> i
[[5, 8, 13], [121, 222, 143, 154]]
>>> j
[[11, 22, 33], [121, 222, 143, 154]]
```

Source: <https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/>



Lists (7/8)

• Deep copies

```
>>> j
[[11, 22, 33], [121, 222, 143, 154]]
>>> import copy
>>> dc = copy.deepcopy(j)
>>> k[1] = 423
>>> j
[[11, 22, 33], [121, 423, 143, 154]]
>>> dc
[[11, 22, 33], [121, 222, 143, 154]]
```

• Membership

```
>>> aList = [1, 2, 3, 5, 0, 9, 0, 7, 1, 5]
>>> 3 in aList
True
>>> 12 in aList
False
```

Source: <https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/>



Lists (8/8)

Deletion

```
>>> aList = [1, 2, 3, 5, 0, 9, 0, 7, 1, 5]
>>> aList.pop(2)
3
>>> aList
[1, 2, 5, 0, 9, 0, 7, 1, 5]
>>> aList.pop(-3)
7
>>> aList
[1, 2, 5, 0, 9, 0, 1, 5]
>>> aList.remove(5)
>>> aList
[1, 2, 0, 9, 0, 1, 5]
>>> del aList[1]
>>> aList
[1, 0, 9, 0, 1, 5]
>>> aList.clear()
>>> aList
[]
```




Tuples (1/5)

- Tuples are ordered, immutable collections of elements, with syntax resembling arrays

```
>>> aTuple=("tuple", 2, 9, [10, 2, 3], "start")
>>> aTuple
('tuple', 2, 9, [10, 2, 3], 'start')
>>> aTuple[0]
'tuple'
>>> aTuple[-1]
'start'
>>> aTuple[2:]
(9, [10, 2, 3], 'start')
>>> aTuple[1] = "tryToChange"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> aTuple.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```



Tuples (2/5)

- Tuples are immutable!

```
>>> anAlias=aTuple
>>> aTuple = aTuple + (1, 2, 3)
>>> aTuple
('tuple', 2, 9, [10, 2, 3], 'start', 1, 2, 3)
>>> anAlias
('tuple', 2, 9, [10, 2, 3], 'start')
>>> aTuple = aTuple + (3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "int") to tuple
>>> aTuple = aTuple + (3,)
>>> aTuple
('tuple', 2, 9, [10, 2, 3], 'start', 1, 2, 3, 3)
>>> len(aTuple)
9
>>> 2 in aTuple
True
>>> 12 not in aTuple
True
```



Tuples (3/5)

- Tuples are immutable ... but be careful of references!

```
>>> a = [3,4,5]
>>> b=(a,6)
>>> b
([3, 4, 5], 6)
>>> a[0]=1
>>> b
([1, 4, 5], 6)
>>> c = b
>>> a[1]=10
>>> b
([1, 10, 5], 6)
>>> c
([1, 10, 5], 6)
>>> c = b.copy()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'copy'
```

Source: https://www.w3schools.com/python/python_tuples.asp/



Tuples (4/5)

- Tuples are immutable ... but be careful of references!

```
>>> import copy
>>> d = copy.deepcopy(b)
>>> d
([1, 10, 5], 6)
>>> a[2]=8
>>> b
([1, 10, 8], 6)
>>> c
([1, 10, 8], 6)
>>> d
([1, 10, 5], 6)
>>>
```

Source: https://www.w3schools.com/python/python_tuples.asp/



Tuples (5/5)

Shortcuts and conversions

```
>>> anAlias=aTuple
>>> aTuple += (55,)
>>> aTuple
('tuple', 2, 9, [10, 2, 3], 'start', 1, 2, 3, 3, 55)
>>> anAlias
('tuple', 2, 9, [10, 2, 3], 'start', 1, 2, 3, 3)
>>> aList = [1, 2, 3]
>>> aConvertedTuple = tuple(aList)
>>> aList
[1, 2, 3]
>>> aConvertedTuple
(1, 2, 3)
>>> aList[2]=5
>>> aList
[1, 2, 5]
>>> aConvertedTuple
(1, 2, 3)
```



Sets (1/3)

- Sets are unordered collections without duplicates whose elements cannot change

```
>>> aSet = {1, 2, "red"}
>>> aSet
{1, 2, 'red'}
>>> anotherSet = {3, 4, "green", 4, "green"}
>>> anotherSet
{3, 4, 'green'}
>>> 1 in aSet
True
>>> "green" in aSet
False
>>> "green" in anotherSet
True
>>> oneIsLikeTrue = { 1, True, "green"}
>>> oneIsLikeTrue
{1, 'green'}
>>> len(oneIsLikeTrue)
2
```



Sets (2/3)

```
>>> aThirdSet = aSet | anotherSet
>>> aThirdSet
{1, 2, 3, 4, 'green', 'red'}
>>> aFourthSet = aThirdSet & {2, 3, 200, 'green'}
>>> aFourthSet
{2, 3, 'green'}
>>>
>>> aSetFromAList = set([9, 8, 7])
>>> aSetFromAList
{8, 9, 7}
>>> aSetFromATuple = set((6, 5, 4))
>>> aSetFromATuple
{4, 5, 6}
>>> aSetFromATuple + aSetFromAList
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

Source: https://www.w3schools.com/python/python_sets.asp/



Sets (3/3)

- Mutable elements like lists, the same sets, and dictionaries cannot be part of sets

```
>>> a = {3, 1, 5, 4}
>>> b = [10, 11, 12]
>>> a.add(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> c = (10, 11, 12)
>>> a.add(c)
>>> a
{1, 3, 4, 5, (10, 11, 12)}
>>> d = {3, 4, 1}
>>> a.add(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

Source: https://www.w3schools.com/python/python_sets.asp/



Dictionaries (1/2)

- Dictionaries are ordered collections of pair (key:value), indexed by key, where each key can appear only once

```
>>> aDictionary = { "breed":"Dog", "weight":8, "name":"Deimon", "age":3}
>>> aDictionary
{'breed': 'Dog', 'weight': 8, 'name': 'Deimon', 'age': 3}
>>> aDictionary["breed"]
'Dog'
>>> len(aDictionary)
4
>>> aDictionary.keys()
dict_keys(['breed', 'weight', 'name', 'age'])
>>> aDictionary.values()
dict_values(['Dog', 8, 'Deimon', 3])
>>> aDictionary.update({"weight":7.5,"color":"blenheim"})
>>> aDictionary
{'breed': 'Dog', 'weight': 7.5, 'name': 'Deimon', 'age': 3, 'color': 'blenheim'}
```

Source: https://www.w3schools.com/python/python_dictionaries.asp/



Dictionaries (2/2)

- Dictionaries can be largely manipulated

```
>>> aDictionary.popitem()
('color', 'blenheim')
>>> aDictionary
{'breed': 'Dog', 'weight': 7.5, 'name': 'Deimon', 'age': 3}
>>> aDictionary.pop('weight')
7.5
>>> aDictionary
{'breed': 'Dog', 'name': 'Deimon', 'age': 3}
>>> del aDictionary["age"]
>>> aDictionary
{'breed': 'Dog', 'name': 'Deimon'}
```

Source: https://www.w3schools.com/python/python_dictionaries_remove.asp/



The range function

- The `range` function is used to generate lists of numbers for iterations
- It returns a range object, which could then be converted in a list for printing

```
>>> list(range(3))
[0, 1, 2]
>>> list(range(-4, 5))
[-4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> list(range(2, 20, 3))
[2, 5, 8, 11, 14, 17]
>>> list(range(2, 20, 7))
[2, 9, 16]
>>> list(range(0))
[]
>>> range(4.3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```



Control flow – if

- The if statement is all based on indentation

```
>>> import math
>>> a = 4
>>> b = 5
>>> c = -1
>>> delta = b**2 - 4*a*c
>>> if delta > 0:
...     x1 = (-b - math.sqrt(delta))/(2*a)
...     x2 = (-b + math.sqrt(delta))/(2*a)
... elif delta == 0:
...     x1 = x2 = - b / (2*a)
... else:
...     x1 = complex(-b, math.sqrt(-delta)/(2*a))
...     x2 = complex(-b, math.sqrt(-delta)/(2*a))
...
>>> x1
-1.425390529679106
>>> x2
0.17539052967910607
>>>
```



Control flow – while

- The `while` statement is the typical top-tested loop based on indentation

```
>>> i = 5
>>> factorial = 1
>>> while i > 1:
...     factorial *= i
...     i -= 1
... else:
...     print("the result is ",factorial)
...
the result is 120
>>>
```

Source: <https://docs.python.org/3/tutorial/controlflow.html/>



Flow in loops

- Python has three constructs to manage the flow in loops:
 - break**: like in C and Java breaks the loop (the **else** statement is not executed)
 - continue**: like in C and Java, suspends the current iteration and jumps directly to the next one
 - pass**: absent in C and Java, moves to the next block
- Now we analyse in details the following snippets

Source: <https://docs.python.org/3/tutorial/controlflow.html/>



break and continue

```
i = 0
print("break")
while i in range(10):
    i += 1
    if i == 5:
        print("i is 5!")
        break
    print("break: I should not get here!")
    print("Standard printout for iteration ",i)
else:
    print("End of loop break")

i = 0
print("continue")
while i in range(10):
    i += 1
    if i == 5:
        print("i is 5!")
        continue
    print("continue: I should not get here!")
    print("Standard printout for iteration ",i)
else:
    print("End of loop continue")
```



pass

```
i = 0
print("pass")
while i in range(10):
    i += 1
    if i == 5:
        print("i is 5!")
        pass
        print("pass: I should not get here!")
    print("Standard printout for iteration ",i)
else:
    print("End of loop pass")
```




Output (1/2)

```
break
Standard printout for iteration 1
Standard printout for iteration 2
Standard printout for iteration 3
Standard printout for iteration 4
i is 5!
continue
Standard printout for iteration 1
Standard printout for iteration 2
Standard printout for iteration 3
Standard printout for iteration 4
i is 5!
Standard printout for iteration 6
Standard printout for iteration 7
Standard printout for iteration 8
Standard printout for iteration 9
Standard printout for iteration 10
End of loop continue
```



Output (2/2)

```
pass
Standard printout for iteration 1
Standard printout for iteration 2
Standard printout for iteration 3
Standard printout for iteration 4
i is 5!
pass: I should not get here!
Standard printout for iteration 5
Standard printout for iteration 6
Standard printout for iteration 7
Standard printout for iteration 8
Standard printout for iteration 9
Standard printout for iteration 10
End of loop pass
```



iterator (1/2)

- This is a case when superclasses are very useful
- `iterator` is a class supporting iterations over collections, that is, referring to a collection of objects, sequentializing and indexing them, and with a method `__next__()` that:
 - returns one by one the elements of the object
 - updates the state of the `iterator` so that it always refers to *next* element in the sequence
 - raises an exception `StopIteration` when there are no more elements to point to
- Please notice the mix of scripting and object orientation

Source: [https://stackoverflow.com/questions/9884132/what-are-iterator-iterable-and-iteration#:~:text=An%20iterable%20is%20a%20object,__next__\(\)%20in%203/](https://stackoverflow.com/questions/9884132/what-are-iterator-iterable-and-iteration#:~:text=An%20iterable%20is%20a%20object,__next__()%20in%203/)



iterator (2/2)

```
>>> string = "iter"
>>> stringIterator = iter(string)
>>> next(stringIterator)
'i'
>>> next(stringIterator)
't'
>>> stringIterator.__next__()
'e'
>>> stringIterator.__next__()
'r'
>>> stringIterator.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Source: [https://stackoverflow.com/questions/9884132/what-are-iterator-iterable-and-iteration#:~:text=An%20iterable%20is%20a%20object,__next__\(\)%20in%203/](https://stackoverflow.com/questions/9884132/what-are-iterator-iterable-and-iteration#:~:text=An%20iterable%20is%20a%20object,__next__()%20in%203/)



iterable

- An `iterable` is the base class for anything that can be iterated over
- It has a method `__iter__()` that returns an `iterator`
- This is also what is done by the global function `iter()` (see slide 52)
- Notice the duality global functions and member functions, like for the global function `next()` and member function `__next__()`
- The duality `iterable` – `iterator` is heavily used in `for` loops, list comprehension etc (see later)

```
>>> string = "iter"
>>> stringIterator = string.__iter__()
>>> next(stringIterator)
'i'
```

Source: <https://stackoverflow.com/questions/9884132/what-are-iterator-iterable-and-iteration>



for loop

- The `for` loop in Python iterates over an `iterator`
- At every step in the loop there is an implicit call to the `__next__()` member function of the iterator
- At the last step there is an implicit catch of the `StopIteration` exception
- `else`, `break`, `continue`, and `pass` work like in a `while` loop

```
>>> string = "iter"
>>> for i in string:
...     print(i)
... else:
...     print("end of string")
...
i
t
e
r
end of string
```



Command line (1/2)

- Command line arguments work like in Java with `argv`
- In addition there is the function `getopt` that helps parsing the arguments one by one

```
import sys, getopt
print("Argument line: ",sys.argv)
print("Number of arguments: ",len(sys.argv))
i=0
for arg in sys.argv:
    print("argument ",i," : ",arg)
    i += 1
options, arguments = getopt.getopt(sys.argv[1:], "nh:o:")
for opt, val in options:
    if opt=="-h" : print ("Help: ",val)
    elif opt=="-o" : print ("Other: ",val)
    elif opt=="-n" : print ("\n means ", end=""); print("\t no arguments")
    else : print(opt, " and ", val, " are not acceptable")
print("The other arguments are ", arguments)
```

Source: <https://docs.python.org/3/library/getopt.html>



Command line (2/2)

- The output is:

```
% python3 commandLine.py -o xxx -h help -n a b c
Argument line: ['commandLine.py', '-o', 'xxx', '-h', 'help', '-n', 'a', 'b', 'c',
']
Number of arguments: 9
argument 0 : commandLine.py
argument 1 : -o
argument 2 : xxx
argument 3 : -h
argument 4 : help
argument 5 : -n
argument 6 : a
argument 7 : b
argument 8 : c
Other: xxx
Help: help
n means no arguments
The other arguments are ['a', 'b', 'c']
```

Source: <https://docs.python.org/3/library/getopt.html>



Python as an imperative language



Functions (1/4)

- In Python there are functions with parameters and return values

```
# exampleFunction.py
import math
def secondOrderEquation(a, b= 3, c= 0.5):
    delta = b*2 -4*a*c
    x1 = (-b - math.sqrt(delta))/(2*a)
    x2 = (-b + math.sqrt(delta))/(2*a)
    return a, b, c, x1, x2

print(secondOrderEquation(1, 6, 1))
print(secondOrderEquation(1, 6))
print(secondOrderEquation(1))
print(secondOrderEquation(1, c=-1))

% python3 exampleFunction.py
(1, 6, 1, -4.414213562373095, -1.5857864376269049)
(1, 6, 0.5, -4.58113883008419, -1.4188611699158102)
(1, 3, 0.5, -2.5, -0.5)
(1, 3, -1, -3.08113883008419, 0.08113883008418976)
```



Functions (2/4)

- Parameters are all passed by values and values can be references, as in Java
- But there are mutable and immutable objects!
- Functions can be parameters of other functions

```
# higherOrder.py
def apply(f,a):
    return f(a)
def increment(x):
    return x+1
def square(x):
    return x**2
def main():
    x = apply(increment,3)
    y = apply(square,4)
    print(x, y)
if __name__ == "__main__":
    main()
% python3 higherOrder.py
4 16
```



Functions (3/4)

- There are anonymous functions
- They are called `lambda` as they resemble lambda expressions

```
# exampleLambda.py
def apply(f,a):
    return f(a)
def main():
    x = apply(lambda x: x+1,3)
    y = apply(lambda x: x**2,4)
    print(x, y)
if __name__ == "__main__":
    main()
% python3 higherOrder.py
4 16
```



Functions (4/4)

- There is a *kind of* `main` function
- Every module has an internal variable, `__name__`
 - this value is set to `__main__` if the module is the one invoked directly in the command line
- Otherwise, it is set to the *name of the module*, that is, the name of the file without the suffix `.py`
- There is a **convention** to call a function `main()` as the first function to be executed by a module that is directly called in the command line

```
if __name__ == "__main__":  
    main()
```



Nested functions

- Functions can be **nested**, like in Pascal

```
# nested.py

def outer(x) :
    y = 10
    def inner(z):
        return z + y
    w = inner(x)
    return w

if __name__ == "__main__":
    print(outer(3))

% python3 nested.py
13
```



Scoping rules (1/5)

- LEGB scoping
 - Local
 - Enclosing
 - Global
 - Builtin
- The keyword `global` declares a variable as global
- The keyword `nonlocal` declares a local variable in an inner scope associated to the outer scope



Scoping rules (2/5)

- Without a **global** or a **nonlocal** declaration:
 - if a variable is initialized with a name already used in an outer scope, then it is treated as a new local variable, which then hides the global one
 - if a variable is used with a name already used in an outer scope, but
 - then** if its value is then changed inside the scope
 - an **error is raised**, as
 - it is treated as a **local variable that previously was used without being initialized**.



Scoping rules (3/5)

```
# scoping.py
x = 5
def testScope(a):
    y = x + a
    global z
    z = y + 1
    return y + z
def main():
    print (testScope(5) + z)
if __name__ == "__main__" :
    main()
% python3 scoping.py
32
```



Scoping rules (4/5)

```
# nonLocalScoping.py
x = 1; y = 2; w = 3; z = 4
def testScope():
    y = 20; w = 30; k = z;
    def testInnerScope():
        w = 300
        global x
        nonlocal y
        x = 100; y = 200; z = 400
        print("testInnerScope: x=",x,"y=",y,"w=",w,"z=",z)
    testInnerScope()
    print("testScope: x=",x,"y=",y,"w=",w,"z=",z)
def main() :
    testScope()
    print("main: x=",x,"y=",y,"w=",w,"z=",z)
if __name__ == "__main__" :
    main()
% python3.11 nonLocalScoping.py
testInnerScope: x= 100 ,y= 200 ,w= 300 ,z= 400
testScope: x= 100 ,y= 200 ,w= 30 ,z= 4
main: x= 100 ,y= 2 ,w= 3 ,z= 4
```



Scoping rules (5/5)

```
# simpleScopingError.py
x = 1
y = 2
z = 3
w = 0
def testScope():
    global x
    y = 4
    z = 5 + w
    global z
    w = -1
def main() :
    print("in main x =",x," y =",y)

global z
~~~~~

SyntaxError: name 'z' is assigned to before global declaration
z = 5 + w
    ^

UnboundLocalError: cannot access local variable 'w' where it is not associated
with a value
```



Default parameters as static (1/3)

- Default parameters are evaluated only once, at the first call
 - they are like static local variables in C
 - normally objects are immutable
 - but when objects are mutable something unexpected may happen



Default parameters as static (2/3)

```
# defaultParametersStatic.py
i = 1
def f(a=[50]):
    global i
    print("Call ", i, "At the beginning of f a is:",a)
    a[0] +=1
    print("At the end of f a is:",a)
    i+=1
def main():
    x = [2]
    print("Before the first call to f x is: ",x)
    f(x)
    print("After the first call to f x is: ",x)
    f()
    print("After the second call to f x is: ",x)
    f()
    print("Before the third call to f x is: ",x)
    f(x)
    print("Before the third call to f x is: ",x)
if __name__=="__main__":
    main()
```



Default parameters as static (3/3)

```
% python3.11 defaultParametersStatic.py
Before the first call to f x is: [2]
Call 1 At the beginning of f a is: [2]
At the end of f a is: [3]
After the first call to f x is: [3]
Call 2 At the beginning of f a is: [50]
At the end of f a is: [51]
After the second call to f x is: [3]
Call 3 At the beginning of f a is: [51]
At the end of f a is: [52]
Before the third call to f x is: [3]
Call 4 At the beginning of f a is: [3]
At the end of f a is: [4]
Before the third call to f x is: [4]
```



Modules (1/3)

- Python structure the code quite like C
- Every file is a module
 - the name of the module is the name of the file without the extension `.py`
 - it is stored in the variable `__name__`
 - such name is changed to `__main__` if the module is the one directly invoked
- A module is an object in Python

Source: <https://docs.python.org/3/tutorial/modules.html>



Modules (2/3)

- To access the module you need to specify the instruction `import amodule`
 - where `amodule.py` is the file where the module is located
 - such file must reside in a location specified by the environmental variable `PYTHONPATH`
 - and any name to use from such module has to be prefixed by the name of the module
 - that is, to use function `goofy()`, the full name `amodule.goofy()` must be specified

Source: <https://docs.python.org/3/tutorial/modules.html>



Modules (3/3)

- To load the names of the module inside the current namespace: `from amodule import goofy`
- To import all the names inside `amodule.py`: `from amodule import *`, in which case only the names starting with an `_` will not be directly accessible
 - in this way it is possible just to call `goofy()`
- **Be careful**, but is also possible to rename an entity or module as
 - `import amodule as unmodulo`, leading to calls like `unmodulo.goofy()` or even
 - `from amodule import goofy as pippo`, leading to calls like `amodule.pippo()`

Source: <https://docs.python.org/3/tutorial/modules.html>



Namespaces

- A namespace is the set of all names associated to objects visible at a certain time and place
- The current namespace is accessible with the instruction `dir()`
- It is also possible to view the namespace of a specific module with `dir(module)`

Source: <https://py-pkgs.org/04-package-structure.html>



Packages (1/2)

- A package is a collection of modules
- From a system viewpoint, a package is a directory:
 - inside the list specified by `sys.path`
 - with a `__init__.py` file, which can also be empty **but must exist**
- From an internal perspective, a package is a module object
- The `import` statement applies for packages like for modules
- A subpackage is a directory of a (sub)packages with a `__init__.py` file
- The `.` separates a package from subpackages and modules

Source: <https://stackoverflow.com/questions/4881897/python-project-and-package-directories-layout> and <https://py-pkgs.org/04-package-structure.html>



Packages (2/2)

- When a package is loaded, its `__init__.py` is executed
- It acts as a kind of constructor of the package
- packages can contain references to other packages using absolute or relative paths
- using relative paths
 - the `.` refers to the current directory
 - the `..` refers to the parent directory

Source: <https://stackoverflow.com/questions/4881897/python-project-and-package-directories-layout> and <https://py-pkgs.org/04-package-structure.html>



Importing elements (1/3)

- `import` is the statement defines the connection between the current module to the one containing the entities we are interested
- when the statement `import A` is executed:
 - the runtime looks for
 - a file named `A.py` (a module) or
 - a directory named `A` with a file `__init__.py` (a packages)
 - in a list of directories specified in the `sys.path` variable
- if the results is a module, then the names inside it become visible
- if the result is a package, then the names inside it become visible and `__init__.py` is executed

Source: <https://chrisyeh96.github.io/2017/08/08/definitive-guide-python-imports.html>



Importing elements (2/3)

- The `import` directive has many variants
 - `import <module or package>`
 - `from <package> import <module or (sub)package or object>`
 - `from <module> import <object>`
- *in Python functions are objects!*
- The structure of the naming after an `import` of **X** is:
 - if **X** is a module or a package:
 - **X.entity**
 - if **X** is a variable:
 - **X** is used as is

Source: <https://chrisyeh96.github.io/2017/08/08/definitive-guide-python-imports.html>



Importing elements (3/3)

- if **X** is a function:
 - the invocation is **X()**
- The imported entity can be renamed placing after the **import X** directive the **as Y**
 - That is:
 - **import <module or package> as Y**
 - **from <package> import <module or (sub)package or object> as Y**
 - **from <module> import <object> as Y**
- From such point on, **Y** must be used instead of **X**

Source: <https://chrisyeh96.github.io/2017/08/08/definitive-guide-python-imports.html>



Python as a functional language

- Already covered
- Lambda expressions
- Some kind of referential transparency
 - Immutable objects
 - Unchangeable (by default) global variables inside local scopes



About overloading of functions (1/3)

- In Python there is not a full overloading of functions as we know it in Java or C++
- If we define multiple times a function, only its latest definition will be used

```
>>> def f(a):  
...     return a  
>>> f(3)  
3  
>>> def f(a,b):  
...     return (a,b)  
>>> f(3,4)  
(3, 4)  
>>> f(3)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: f() missing 1 required positional argument: 'b'
```

Source: <https://www.geeksforgeeks.org/python-method-overloading/>



About overloading of functions (2/3)

- The standard way to do it is via classes (see later)
- Default parameters may help
- They can be coupled with a defaulting to None
- Or we can use “decorators” (also explained later) **but with care**
- To use the required decorator we first need to install the suitable package `multipledispatch`

```
% pip3 install multipledispatch
Collecting multipledispatch
Downloading multipledispatch-1.0.0-py3-none-any.whl.metadata (3.8 kB)
Downloading multipledispatch-1.0.0-py3-none-any.whl (12 kB)
Installing collected packages: multipledispatch
Successfully installed multipledispatch-1.0.0
```

Source: <https://www.geeksforgeeks.org/python-method-overloading/>



About overloading of functions (3/3)

```
>>> from multipledispatch import dispatch
>>> @dispatch(int)
... def f(a):
...     return a
>>> @dispatch(int, int)
... def f(a,b):
...     return a+b
>>> f(3,4)
7
>>> f(3)
3
>>> def f(a,b):
...     return a-b
>>> f(3,4)
-1
>>> f(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() missing 1 required positional argument: 'b'
```

Source: <https://www.geeksforgeeks.org/python-method-overloading/>



Functions are first class citizens

```
>>> def aFunction(a):
...     return a+1
>>> aFunction(3)
4
>>> x = aFunction
>>> x(3)
4
>>> l = [x, aFunction, 88, "bye"]
>>> l[0]
<function aFunction at 0x100ff0d60>
>>> l[1]
<function aFunction at 0x100ff0d60>
>>> l[2]
88
>>> l[3]
'bye'
>>> type(x)
<class 'function'>
```

Source: <https://realpython.com/python-functional-programming/>



Functions as parameters

- There are also predefined functions for this, such as `map`, `filter`, and `reduce`, which are used with the *map-reduce* pattern present in big data
- `map` returns an iterator

```
>>> def apply(f,x):  
...     return f(x)  
>>> def incr(x):  
...     return x+1  
...  
>>> apply(incr,3)  
4  
>>> x = map(incr,[1,2,3,4])  
>>> x  
<map object at 0x100b9bf40>  
>>> list(x)  
[2, 3, 4, 5]
```

Source: <https://realpython.com/python-functional-programming/>



map plays a pivotal role

- It may take a different form using polymorphism
- Notice the polymorphic structure of the operator +
- `map` can be computed in constant time if there are enough processors, also taking advantage of a data-parallel approach

```
>>> def f(a,b,c):  
...     return a+b+c  
>>> f(1,2,3)  
6  
>>> f("a","b","c")  
'abc'  
>>> list(map(f,["a","b","c","d"],["e","f","g","h"],["i","j","k","l"])))  
['aei', 'bfj', 'cgk', 'dhl']  
>>>
```

Source: <https://realpython.com/python-functional-programming/>



filter selects elements

- The elements of a `filter` can be selected via the `filter`
- Inside `filter` the lambda functions are particularly handy
- `filter` returns an iterator
- Also `filter` can be computed in constant time if there are enough processors

```
>>> def f(a):  
...     return a>10  
>>> f(3)  
False  
>>> f(11)  
True  
>>> filter(f, [8,9,10,3,11,2,100,-11])  
<filter object at 0x1006815d0>  
>>> list(filter(f, [8,9,10,3,11,2,100,-11]))  
[11, 100]  
>>> list(filter(lambda x: x>10, [8,9,10,3,11,2,100,-11]))  
[11, 100]  
>>>
```



reduce synthesises elements

- The elements of an iterator can be folded in one with **reduce**
- They can be the result of a **map** and/or a **filter**
- The function for the reduction must have two elements, where the second must be of the type of the elements of the iterator
- **reduce** can be computed in log time if there are enough processors

```
>>> from functools import reduce
>>> def f(a,b):
...     return a+b
...
>>> reduce(f,[1,2,3,4,5])
15
>>> reduce(lambda x,y: x+y,[1,2,3,4,5])
15
>>> reduce(lambda x,y: x+y,map(lambda x: x**2,filter(lambda x: x %
2!=0,[1,2,3,4,5])))
35
>>>
```




Lazy evaluation (1/3)

- Python has an eager evaluation of parameters passed to functions
- Python provides some form approximating lazy evaluation
- There are objects supplying a large stream of data one element at a time *on demand*, using the statement **yield** instead of the usual **return**
- This is very similar on how the elements of a list are accessed via iteration
- They can also be expanded in other data structures, like tuples
- Note that, once expanded, they are completely consumed, that is, lost

Source: <https://docs.python.org/3/howto/functional.html>



Lazy evaluation (2/3)

```
>>> l = [2,4,6,8,10,12]
>>> y = iter(l)
>>> y
<list_iterator object at 0x100681390>
>>> next(y)
2
>>> next(y)
4
>>> tuple(y)
(6, 8, 10, 12)
>>> tuple(y)
()
```

Source: <https://docs.python.org/3/howto/functional.html>



Lazy evaluation (3/3)

```
>>> def generateIntegers(N):  
...     for i in range(N):  
...         yield i  
...  
>>> x = generateIntegers(10)  
>>> x  
<generator object generateIntegers at 0x1002caf60>  
>>> next(x)  
0  
>>> next(x)  
1  
>>> t = tuple(x)  
>>> t  
(2, 3, 4, 5, 6, 7, 8, 9)  
>>> tuple(x)  
()  
>>> t  
(2, 3, 4, 5, 6, 7, 8, 9)
```

Source: <https://docs.python.org/3/howto/functional.html>



List comprehension

- Python has very effective means to generate lists, as several functional languages; such means are collectively referred to with the term redlist comprehension
- Iterators are at the base of list comprehension
- List comprehension is heavily used in Python, as it is a very compact and expressive

```
>>> [x for x in range(10)]  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
>>> [x*x for x in (filter(lambda l: l%2==0,range(5,12)))]  
[36, 64, 100]  
>>> [x*x for x in range(5,12) if x%2==0]  
[36, 64, 100]  
>>> [x*x if x%2==0 else -x*x for x in range(5,12)]  
[-25, 36, -49, 64, -81, 100, -121]
```

Source: <https://realpython.com/list-comprehension-python/> and
https://www.w3schools.com/python/python_lists_comprehension.asp



Beyond List comprehension

- There are also *set comprehension* and *dictionary comprehension*
 - dictionary comprehension requires the pairs with :
- we could have nested or *zip*-ed comprehension
 - zip* returns an *iterator* of tuples

```
>>> [x**2 for x in range(-3,3)]
[9, 4, 1, 0, 1, 4]
>>> {x**2 for x in range(-3,3)}
{0, 9, 4, 1}
>>> [(i,j) for i in range(3) for j in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
>>> [(i,j) for (i,j) in zip(range(3),range(3))]
[(0, 0), (1, 1), (2, 2)]
>>> {i:j[::-1] for (i,j) in zip(range(1,4),["one","two","three"])}
{1: 'eno', 2: 'owt', 3: 'eerht'}
```

Source: <https://www.freecodecamp.org/news/dictionary-comprehension-in-python-explained-with-examples/> and https://www.w3schools.com/python/ref_func_zip.asp



Python as an object oriented language (1/5)

- Python is an object oriented language (to a certain extent)
- Classes are defined with `class` and with the usual indentation
- The reference to the object itself is `self`
- Instance methods requires are identified by having as parameter `self`, which is then omitted when calling
- The initializer (a kind of constructor) is `__init__`
- Instance variables are *usually* defined in the constructor

```
class Animal:
    def __init__(self):
        self.name=""
    def setName(self,name):
        self.name=name
if __name__ == '__main__':
    animal = Animal()
    animal.setName("Pippo")
```



Python as an object oriented language (2/5)

- Class variables are defined globally in the class
- Class methods are defined with the `@classmethod` *decoration*, with a reference to the class, `cls`, and with the usual indentation

```
class Animal:
    numberOfAnimals = 0
    def __init__(self):
        Animal.numberOfAnimals += 1
        self.name=""
    def setName(self,name):
        self.name=name
    @classmethod
    def getNumberOfAnimals(cls):
        return cls.numberOfAnimals
if __name__ == '__main__':
    animal = Animal()
    animal.setName("Pippo")
    i = Animal.getNumberOfAnimals()
```



Python as an object oriented language (3/5)

- There are static methods non containing any reference to the class and defined with the `@staticmethod` decoration
 - Static methods exist only for naming reasons
- Protected methods and data start with the single underscore `_`
- Private methods and data start with the double underscore `__`
- However, both can be accessed outside the class with *name mangling*, that is, prefixing its name with `_ClassName`

```
class Animal:
    numberOfAnimals = 0
    def __init__(self, name):
        Animal.numberOfAnimals += 1
        self.name=name
        self.__private = 1
        self._protected = 1
        self.public = 1
```

Source: <https://www.geeksforgeeks.org/private-methods-in-python/> and
<https://jellis18.github.io/post/2022-01-15-access-modifiers-python/>



Python as an object oriented language (4/5)

```
def identifyYourself(self):  
    print("Hello! I am a generic animal and my name is " + self.name)  
  
def __privateChangeName(self, name):  
    self.name = name  
  
@classmethod  
def getNumberOfAnimals(cls):  
    return cls.numberOfAnimals  
  
@staticmethod  
def increment(i):  
    return i+1
```



Python as an object oriented language (5/5)

```
if __name__ == '__main__':  
    animal = Animal("Mystery")  
    animal.identifyYourself()  
    # animal.__privateChangeName("pippo")  
    animal._Animal__privateChangeName("MoreMystery")  
    i = Animal.getNumberOfAnimals()  
    i = Animal.increment(i)  
    animal._Animal_protected = i  
    print(animal._Animal_protected)  
    print(animal.name)  
2  
MoreMystery
```



Inheritance in Python

- Python supports single and multiple inheritance
- Every class without a superclass is implicitly derived from the class `object`
- `object` is then of type of `type`
- When a new object is created the initialized method `__init__` is called
- Here below a summary of a derived class

```
class Dog(Animal):  
  
    def __init__(self, name):  
        super().__init__(name)  
  
    def identifyYourself(self):  
        print("Hello! I am a dog and my name is " + self.name)
```



About `__init__` (1/2)

- Remember that `__init__` is an initializer, therefore, by default the creation of a derived class does not trigger the `__init__` of the base class
 - If an initialization from the superclass is needed, it needs to be explicitly called explicitly
- However, if there is no `__init__` in the derived class, the system will look for the closest one going up the hierarchy, since it is a virtual function

```
class Animal:
    def __init__(self):
        print("Creating an animal")

class Penguin(Animal):
    def __init__(self):
        print("Creating a penguin")
```



About `__init__` (2/2)

```
class Parrot(Animal):
    def __init__(self):
        super().__init__()
        print("Creating a parrot")

class Dove(Animal):
    pass

if __name__ == "__main__":
    a = Animal()
    pe = Penguin()
    pa = Parrot()
    d = Dove()
```

- And therefore the output is:

```
Creating an animal
Creating a penguin
Creating an animal
Creating a parrot
Creating an animal
```



Operator overloading (1/5)

- A special kind of such member functions are those supporting operator overloading
- There is a specific mapping between names of functions and operator that is overloaded
- Such functions are “special functions” or “double underscore functions” or “magic methods”, such as `__add__()`

```
class Animal:

    def __add__(self, anotherAnimal):
        return Animal(self.name+anotherAnimal.name)

if __name__ == '__main__':
    animal = Animal("Mystery")
    anotherAnimal = Animal("BigMystery")
    thirdAnimal = animal + anotherAnimal
```



Operator overloading (2/5)

- A list of the standard mapping of names and operator is presented in this and the following slide

+	A + B	<code>__add__(self, other)</code>
-	A - B	<code>__sub__(self, other)</code>
*	A * B	<code>__mul__(self, other)</code>
/	A / B	<code>__truediv__(self, other)</code>
//	A // B	<code>__floordiv__(self, other)</code>
%	A % B	<code>__mod__(self, other)</code>
**	A ** B	<code>__pow__(self, other)</code>
>>	A >> B	<code>__rshift__(self, other)</code>
<<	A << B	<code>__lshift__(self, other)</code>
&	A & B	<code>__and__(self, other)</code>
	A B	<code>__or__(self, other)</code>
^	A ^ B	<code>__xor__(self, other)</code>
<	A < B	<code>__lt__(SELF, OTHER)</code>
>	A > B	<code>__gt__(SELF, OTHER)</code>
<=	A <= B	<code>__le__(SELF, OTHER)</code>

Source of the picture

<https://faun.pub/the-right-way-to-overload-methods-and-operators-in-python-2f93232af031/>



Operator overloading (3/5)

<code><=</code>	<code>A <= B</code>	<code>__LE__(SELF, OTHER)</code>
<code>>=</code>	<code>A >= B</code>	<code>__GE__(SELF, OTHER)</code>
<code>==</code>	<code>A == B</code>	<code>__EQ__(SELF, OTHER)</code>
<code>!=</code>	<code>A != B</code>	<code>__NE__(SELF, OTHER)</code>
<code>-=</code>	<code>A -= B</code>	<code>__ISUB__(SELF, OTHER)</code>
<code>+=</code>	<code>A += B</code>	<code>__IADD__(SELF, OTHER)</code>
<code>*=</code>	<code>A *= B</code>	<code>__IMUL__(SELF, OTHER)</code>
<code>/=</code>	<code>A /= B</code>	<code>__IDIV__(SELF, OTHER)</code>
<code>//=</code>	<code>A //= B</code>	<code>__IFLOORDIV__(SELF, OTHER)</code>
<code>%=</code>	<code>A %= B</code>	<code>__IMOD__(SELF, OTHER)</code>
<code>**=</code>	<code>A **= B</code>	<code>__IPOW__(SELF, OTHER)</code>
<code>>>=</code>	<code>A >>= B</code>	<code>__IRSHIFT__(SELF, OTHER)</code>
<code><<=</code>	<code>A <<= B</code>	<code>__ILSHIFT__(SELF, OTHER)</code>
<code>&=</code>	<code>A &= B</code>	<code>__IAND__(SELF, OTHER)</code>

Source of the picture

<https://faun.pub/the-right-way-to-overload-methods-and-operators-in-python-2f93232af031/>



Operator overloading (4/5)

- There are additional “double underscore” functions mimicking the structure of C++, such as
 - `__getitem__(self, index)` for subscripting in the rhs, that is the `x = anObject[index]`
 - `__setitem__(self, index)` for subscripting in the lhs, that is the `anObject[index] = x`
 - `__contains__(self, index)` for the `in` operator
 - `__repr__(self)` to convert an object in a string, used in `print` with the template pattern
 - `__iter__(self)` to generate an iterable, like `iter(anObject)` to use then, say, in a `for`
 - `__enter__(self)` and `__exit__(self, ...)` to handle entering and exiting blocks

<https://www.analyticsvidhya.com/blog/2021/08/explore-the-magic-methods-in-python/>



Operator overloading (5/5)

- `__call__(self, whatever)` for *functional objects*, that is for managing structures like `anObject(whatever)`
 - It is also used to create objects as discussed later
- `__new__(...)` for allocating space for objects
- `__init__(...)` for initializing objects

<https://www.analyticsvidhya.com/blog/2021/08/explore-the-magic-methods-in-python/>



Understanding the object in Python (1/2)

- In Python an object has:
 - an identity
 - a value or a state, defined by the values of its attributes
 - a type
 - one or more bases, something similar to a superclass

```
>>> type(3)
<class 'int'>
>>> type(Animal)
<class '__main__.Animal'>
>>> type(dog)
<class '__main__.Dog'>
>>> type(int)
<class 'type'>
>>> type(Animal)
<class 'type'>
>>> type(Dog)
<class 'type'>
```

Source: Shalabh Chaturvedi "Python Types and Objects"
<https://www.eecg.toronto.edu/~jzhu/csc326/readings/metaclass-class-instance.pdf/>



Understanding the object in Python (2/2)

- Also the object has a type
- And also the type!

```
>>> type(object)
<class 'type'>

>>> type(type)
<class 'type'>
```

Source: Shalabh Chaturvedi “Python Types and Objects”
<https://www.eecg.toronto.edu/~jzhu/csc326/readings/metaclass-class-instance.pdf/>



The notion of object (1/4)

- There is some duality to explore between type and base class, which requires some deepening
- First of all, we notice the reflection primitives, `type` and `dir`

```
>>> dir(3)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '
__delattr__', '__dir__', ...]
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '
__delattr__', '__dir__', ...]
>>> dir(animal)
['_Animal__private', '_Animal__privateChangeName', '_Animal__protected', '
__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '
__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '
__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__
', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '
__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '
_protected', 'getNumberOfAnimals', 'identifyYourself', 'increment', 'name'
, 'public']
```

Source: Shalabh Chaturvedi “Python Types and Objects”

<https://www.eecg.toronto.edu/~jzhu/csc326/readings/metaclass-class-instance.pdf/>



The notion of object (2/4)

```
>>> dir(dog)
['_Animal__private', '_Animal__privateChangeName', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_protected', 'getNumberOfAnimals', 'identifyYourself', 'increment', 'name', 'numberOfAnimals', 'public']
```

- To rationalize our understanding we need to explore the meaning of the attributes `__base__` and `__class__`
- We will use reflection again

Source: Shalabh Chaturvedi “Python Types and Objects”
<https://www.eecg.toronto.edu/~jzhu/csc326/readings/metaclass-class-instance.pdf/>



The notion of object (3/4)

```
>>> animal.__class__
<class '__main__.Animal'>
>>> type(animal.__class__)
<class 'type'>
>>> animal.__class__.__base__
<class 'object'>
>>> type(animal.__class__.__base__)
<class 'type'>

>>> dog.__class__
<class '__main__.Dog'>
>>> type(dog.__class__)
<class 'type'>
>>> dog.__class__.__base__
<class '__main__.Animal'>
>>> type(dog.__class__.__base__)
<class 'type'>
```

Source: Shalabh Chaturvedi “Python Types and Objects”
<https://www.eecg.toronto.edu/~jzhu/csc326/readings/metaclass-class-instance.pdf/>



The notion of object (4/4)

- Now we can explore the objects `object` and `type`

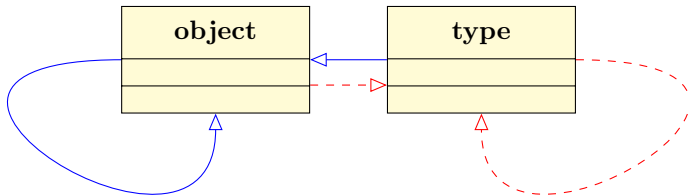
```
>>> object.__class__
<class 'type'>
>>> type(object.__class__)
<class 'type'>
>>> object.__class__.__base__
<class 'object'>
>>> type(object.__class__.__base__)
<class 'type'>

>>> type.__class__
<class 'type'>
>>> type(type.__class__)
<class 'type'>
>>> type.__class__.__base__
<class 'object'>
>>> type(type.__class__.__base__)
<class 'type'>
```

Source: Shalabh Chaturvedi "Python Types and Objects"
<https://www.eecg.toronto.edu/~jzhu/csc326/readings/metaclass-class-instance.pdf/>



A view of object and type in UML





Revising our objects (1/5)

- (Class) `Animal`
 - identity (`Animal.__name__`) : `Animal`
 - a type (`Animal.__class__`) : `type`
 - a base (`Animal.__base__`) : `object`
- (Object) `animal`
 - there is no attribute `animal.__name__`
 - a type (`Animal.__class__`) : `__main__.Animal`
 - there is no attribute `animal.__base__`



Revising our objects (2/5)

- (Class) `Dog`
 - identity (`Dog.__name__`) : `Dog`
 - a type (`Dog.__class__`) : `type`
 - a base (`Dog.__base__`) : `__main__.Animal`
- (Object) `dog`
 - there is no attribute `dog.__name__`
 - a type (`dog.__class__`) : `__main__.Dog`
 - there is no attribute `dog.__base__`



Revising our objects (3/5)

- *(Object)* object
 - identity (`object.__name__`) : `object`
 - a type (`object.__class__`) : `type`
 - a base (`object.__base__`) : `None`
- *(Class)* object
 - identity (`object.__class__.__name__`) : `type`
 - a type (`object.__class__.__class__`) : `type`
 - a base (`object.__class__.__base__`) : `None`



Revising our objects (4/5)

- (*Base class of object*) aka `object.__base__` `None`
 - there is no attribute `object.__base__.__name__`
 - a type (`object.__base__.__class__`) : `NoneType`
 - there is no attribute `object.__base__.__base__`
- (*Type of the base class of object*) aka `None.__base__` aka `object.__base__.__class__` : `NoneType`
 - identity (`object.__base__.__class__.__name__`) : `NoneType`
 - type (`object.__base__.__class__.__class__`) : `type`
 - there is no attribute `object.__base__.__class__.__base__`



Revising our objects (5/5)

- (*Object*) `type`
 - identity (`type.__name__`) : `type`
 - a type (`type.__class__`) : `type`
 - a base (`type.__base__`) : `object`
- (*Class of*) `type`
 - identity (`object.__class__.__name__`) : `object`
 - a type (`object.__class__.__class__`) : `type`
 - a base (`object.__class__.__base__`) : `type`
 - It is the `object`, but with some inconsistency!!



Putting the pieces together

- We have the structure of the Abstract Factory
- Everything is (derived of) an `object`
- Every object has a `type`
- A `type` is an object and derived (also indirectly) from `object`
- The `type` is identified by the `__base__` attribute
- The base class is identified by the `__class__` attribute
- A metaclass is a class whose instances are still classes
- Metaclasses play an important role in the creation/instantiation process

Source: Shalabh Chaturvedi "Python Types and Objects"
<https://www.eecg.toronto.edu/~jzhu/csc326/readings/metaclass-class-instance.pdf/>



Instantiation of new objects

- The reference for creating new objects is the clone design pattern
- New objects can be created by subclassing

```
class Dog(Animal):  
  
    def __init__(self, name):  
        super().__init__(name)
```

new object of type `type` has been instantiated with base class object (the full specs in slide 115) New objects can be created by applying the operator `()` to an object of type `type`

```
dog = Dog()  
o = object()  
animal = Animal()
```

Source: Shalabh Chaturvedi "Python Types and Objects"
<https://www.eecg.toronto.edu/~jzhu/csc326/readings/metaclass-class-instance.pdf/>



Instantiation of new types (1/2)

- Types can be created on the fly
- Extreme care is required

```
>>> Cat = type("Cat", (Animal,), {"nickname": "Prr", "weight": 10})
>>> c = Cat("Garfield")
>>> print(c)
<__main__.Cat object at 0x11148fa90>
>>> print(c.nickname)
Prr
>>> print(c.weight)
10
>>> print(c.name)
Garfield
>>> print(c.somethingElse)
print(c.somethingElse)
~~~~~
AttributeError: 'Cat' object has no attribute 'somethingElse'
>>> c.somethingElse=2
>>> print(c.somethingElse)
2
```

Source: <https://realphysics.info/Theory%20of%20Python/classes.html/>



Instantiation of new types (2/2)

- Types can be returned from functions

```
def generateCongruent(something):  
    return type(something)  
t = generateCongruent(c)  
x = t("Garfield2, the revenge")  
type(x)  
Cat    Garfield2, the revenge  
type(x.something)  
^^^^^^^^  
AttributeError: 'Cat' object has no attribute 'something'
```

Source: <https://realphysics.info/Theory%20of%20Python/classes.html/>



The creation process (1/2)

- The creation of an object takes two (plus one) steps:
 - `__new__()`, in principle to allocate memory
 - `__init__()`, in principle to initialize data
- The original object is taken as a reference
- It is also possible to modify the creation of objects using **metaclasses**
 - Using metaclasses it is possible to alter the creation process of objects
 - In this case, a rigid protocol is employed, using the template pattern, again, based on overriding / late binding

Source: Shalabh Chaturvedi "Python Types and Objects"
<https://www.eecg.toronto.edu/~jzhu/csc326/readings/metaclass-class-instance.pdf/>,
<https://www.datacamp.com/tutorial/python-metaclasses>, and
<https://stackoverflow.com/questions/56514586/arguments-of-new-and-init-for-metaclasses>



The creation process (2/2)

- Overall we see three methods clearly involved in the protocol
 - `__call__()`, the orchestrator, called every time an object type is invoked to create a new object, like in

```
dog = Dog()
```

- `__new__()`, called to allocate memory, as mentioned
 - `__init__()`, called to initialize the data, as mentioned
- `__call__()` calls the other two, therefore imposing a structure on the number and types of parameters
- this is why we may see typing error if they are not properly structured

Source: Shalabh Chaturvedi "Python Types and Objects"

<https://www.eecg.toronto.edu/~jzhu/csc326/readings/metaclass-class-instance.pdf/>,

<https://www.datacamp.com/tutorial/python-metaclasses>, and

<https://stackoverflow.com/questions/56514586/arguments-of-new-and-init-for-metaclasses>



Inside the creation (1/5)

- The creation process involves metaclasses
- `type` is acting as the “metametaaclass”
- `*` and `**` allow arbitrary number of parameters:
 - `*` implies that the argument (here, `args`) is a tuple
 - `**` implies that the argument (here, `kw`) is a dictionary
 - Here we see the equivalent in Python of the real C code (taken from the reference below)

```
class type:
    def __call__(cls, *args, **kw):
        instance = cls.__new__(cls, *args, **kw)
        # __new__ is actually a static method -
        # cls has to be passed explicitly
        if isinstance(instance, cls):
            instance.__init__(*args, **kw)
        return instance
```

Source: <https://www.datacamp.com/tutorial/python-metaclasses>, and
<https://stackoverflow.com/questions/56514586/arguments-of-new-and-init-for-metaclasses>



Inside the creation (2/5)

- Substantially `__new__()` and `__init__()` ought to be redefined
- Let's see an implementation of the singleton pattern

```
class Singleton:
    singleObject = None
    @staticmethod
    def __new__(cls):
        if Singleton.singleObject is not None:
            return Singleton.singleObject
        else:
            Singleton.singleObject = super().__new__(cls)
            return Singleton.singleObject

    def __init__(self):
        self.aValue = 1

    def increment(self):
        self.aValue += 1
```



Inside the creation (3/5)

- What is the output?

```
>>> if __name__ == "__main__":  
>>>     x = Singleton()  
>>>     print(x.aValue)  
>>>     x.increment()  
>>>     print(x.aValue)  
>>>     y = Singleton()  
>>>     print(x.aValue)  
>>>     print(y.aValue)  
>>>     print(x==y)
```



Inside the creation (4/5)

```
>>> if __name__ == "__main__":  
>>>     x = Singleton()  
>>>     print(x.aValue)  
1  
>>>     x.increment()  
>>>     print(x.aValue)  
2  
>>>     y = Singleton()  
>>>     print(x.aValue)  
1  
>>>     print(y.aValue)  
1  
>>>     print(x==y)  
True
```

- `__init__()` is still called!!
- We need some refinement
 - Remember that for `type` we cannot modify the sequence of calling first `__init__()` and then `__new__()`
 - it is built inside the language



Inside the creation (5/5)

```
def __init__(self):
    if Singleton.numberOfItems == 0:
        Singleton.numberOfItems = 1
        self.aValue = 1

>>> if __name__ == "__main__":
>>>     x = Singleton()
>>>     print(x.aValue)
1
>>>     x.increment()
>>>     print(x.aValue)
2
>>>     y = Singleton()
>>>     print(x.aValue)
2
>>>     print(y.aValue)
2
>>>     print(x==y)
True
```



Metaclasses (1/9)

- Remember that in Python everything is an object, including classes
- As mentioned,
 - a metaclass is a class whose instances are still classes
 - the creation process involves metaclasses
 - `type` is acting as the “metametaclass”
- The instruction

```
class Animal:  
    def __init__(self):  
        print("Creating an animal")
```

results in the creation (instantiation) of:

- a `class Animal` implicitly derived from `class object`
- an `object Animal` instance of the `class type`

Source: <https://www.datacamp.com/tutorial/python-metaclasses>



Metaclasses (2/9)

- Moreover, with this code

```
animal = Animal()
```

the object `Animal` creates an instance of class `Animal` and returns its value to `animal`

- This is clearly an use of the method `__call__()`
- So the question is whether it is possible for the class `Animal` to produce also other classes
- This is the core of metaclasses

Source: <https://www.datacamp.com/tutorial/python-metaclasses>



Metaclasses (3/9)

- We start with defining a subclass of **class type**

```
class AnimalType(type):  
    pass
```

- As usual here we create
 - a **class AnimalType** explicitly derived from **class type**
 - an **object AnimalType** instance of the **class type**

Source: <https://www.datacamp.com/tutorial/python-metaclasses>



Metaclasses (4/9)

- Then we use such class to proceed

```
class Animal(metaclass=AnimalType):  
    pass
```

- And here we create
 - a **class Animal** with:
 - meta-class **class AnimalType**, meaning that its associated object is instance of **class AnimalType**, and
 - implicitly derived from **class object**
 - an **object Animal** instance of the **class AnimalType**, as mentioned above

Source: <https://www.datacamp.com/tutorial/python-metaclasses>



Metaclasses (5/9)

- Metaclassing allows the manipulation of the class as a whole, especially its creation process
- We now consider an example

```
class AnimalType(type):
    def __init__(cls, name, bases, dct):
        print("__init__ of AnimalType")
    def __call__(self, *args, **kwargs):
        print("__call__ of AnimalType")
        return super().__call__(self, *args, **kwargs)
    def __new__(cls, name, *args, **kwargs):
        print("__new__ of AnimalType")
        return super().__new__(cls, name, *args, **kwargs)

class Animal(metaclass=AnimalType):
    def __init__(self, *args):
        print("__init__ of Animal")
    def __new__(cls, *args, **kwargs):
        print("__new__ of Animal")
        return super().__new__(cls)
```

Source: <https://www.datacamp.com/tutorial/python-metaclasses>



Metaclasses (6/9)

```
__new__ of AnimalType # To create the object AnimalType at the beginning of the
                        module
__init__ of AnimalType # To initialize the object AnimalType
>>> if __name__ == '__main__':
>>>     print("Now creating an Animal")
Now creating an Animal
>>>     a = Animal()
__call__ of AnimalType # This is the result of the call Animal(
__new__ of Animal # Called by AnimalType.__call__
__init__ of Animal # Called by AnimalType.__call__
```

- The `__call__()` of the metaclass (in our example `class AnimalType`) orchestrates the overall creation process of the objects of `class Animal`
 - It is an application of the *Template Method* design pattern
- This allows a fine-grained control of the creation process

Source: <https://www.datacamp.com/tutorial/python-metaclasses>



Metaclasses (7/9)

- The `__call__()` of the metaclass calls of the target class `__new__()` and `__init__()`
- Therefore, the metaclass has its own:
 - `__call__(cls, *args, **kwargs)`, called when an instance of the metaclass is called, for instance when creating an object of the class, like in `a = Animal()`
 - `cls` is the reference to the metaclass
 - `args` is the list of positional arguments
 - `kwargs` is the list of the keyword arguments

Source:

<https://www.datasciencecentral.com/understanding-the-complexity-of-metaclasses-and-their-practical-2/>



Metaclasses (8/9)

- ... therefore, the metaclass has its own:
 - `__new__(mcs, name, bases, namespace)`, where
 - `mcs` is the reference to the metaclass
 - `name` is the name to the metaclass
 - `bases` is the list of the superclasses, which will become the `__base__` attribute of the new class
 - `namespace` is the namespace in the form of a dictionary, which will become the `__dict__` attribute of the new class

Source:

<https://www.datasciencecentral.com/understanding-the-complexity-of-metaclasses-and-their-practical-2/>



Metaclasses (9/9)

- ... therefore, the metaclass has its own:
 - `__init__(cls, name, bases, namespace, **kwargs)`
 - `mcs` is the reference to the metaclass
 - `name` is the name to the metaclass
 - `bases` is the list of the superclasses, which will become the `__base__` attribute of the new class
 - `namespace` is the namespace in the form of a dictionary, which will become the `__dict__` attribute of the new class
 - `kwargs` is the list of the keyword arguments
 - There is also a `__prepare__()` method, called as the first method to prepare the namespace

Source:

<https://www.datasciencecentral.com/understanding-the-complexity-of-metaclasses-and-their-practical-2/>



Singleton with metaclasses (1/4)

- We can use it for an alternative implementation of the Singleton pattern

```
class AnimalType(type):
    singleAnimal=None
    def __init__(cls, name, bases, dct):
        print("__init__ of AnimalType")

    def __call__(cls, *args, **kwargs):
        print("__call__ of AnimalType")
        if AnimalType.singleAnimal is None:
            print("Creating the unique animal")
            AnimalType.singleAnimal = cls.__new__(cls, *args, **kwargs)
            AnimalType.singleAnimal.__init__(*args, **kwargs)
        else:
            print("The unique animal already exists")
        return AnimalType.singleAnimal

    def __new__(cls, name, *args, **kwargs):
        print("__new__ of AnimalType")
        return super().__new__(cls, *args, **kwargs)
```



Singleton with metaclasses (2/4)

```
class Animal(metaclass=AnimalType):
    def __init__(self,*args):
        print("__init__ of Animal")
        print(f'self = {self}; args = {args}')
        self.weight=10
        print(f'self.weight = {self.weight}; args = {args}')

    def __new__(cls, *args, **kwargs):
        print("__new__ of Animal")
        if AnimalType.singleAnimal is None:
            print("Creating the unique singleton")
            AnimalType.singleAnimal = super().__new__(cls, *args, **kwargs)
        return AnimalType.singleAnimal
```



Singleton with metaclasses (3/4)

- Now we start the execution

```
__new__ of AnimalType # The object AnimalType is created
__init__ of AnimalType
>>> if __name__ == '__main__':
>>>     print("Now creating an Animal")
Now creating an Animal
>>>     print(type(AnimalType))
<class 'type'>
>>>     print(type(Animal))
<class '__main__.AnimalType'>
>>>     a = Animal() # This is a call to the singleton;
>>>.      # there is no need of a method called getSingleton
__call__ of AnimalType
Creating the unique animal
__new__ of Animal
Creating the unique singleton
__init__ of Animal
```



Singleton with metaclasses (4/4)

```
>>> print(a)
<__main__.Animal object at 0x106f24ed0>
>>> print(a.weight)
10
>>> a.weight = 20
>>> print(a.weight)
20
>>> b = Animal() # Having overloaded AnimalType.__call__(),
>>>. # no new object is created but the singleton is returned
The unique animal already exists
>>> print(b)
<__main__.Animal object at 0x106f24ed0>
>>> print(a==b)
True
>>> b.weight = 300
>>> print(a.weight)
300
```



Generalization of Singleton (1/2)

- It is possible to create a general meta class to use when we need a singleton of any class

```
class Singleton(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls]=super(type(cls), cls).__call__(*args, **kwargs)
        return cls._instances[cls]
```

- Here we let `__call__` invoke the `__call__` of the superclass only if no instances of `cls` have already been created

Source: <https://itnext.io/deciding-the-best-singleton-approach-in-python-65c61e90cdc4>



Generalization of Singleton (2/2)

- Now suppose that we want to create a specialization of a Database class allowing only an instance of it

```
class Database():
    def __init__(self, url):
        self.url = url

    def connect(self):
        if 'https' not in self.url:
            raise ValueError("invalid url: it must be encrypted")
        return True

class DatabaseSingleton(Database, metaclass=Singleton):
    pass
```

- Specifying as metaclass **Singleton** we have achieved our goal

Source: <https://itnext.io/deciding-the-best-singleton-approach-in-python-65c61e90cdc4>



Questions?

End of the introductory lectures on Python.