Python Cert Prep Latex Config for org-mode

shae128

December 16, 2018

Contents

1 Introduction

- A complete set of known commands is called an -instruction list-, sometimes abbreviated to **IL**.
- First of all, the interpreter checks if all subsequent lines are correct. If the compiler finds an error, **it finishes its work immediately**.
- each line is usually executed separately, so the trio "read-check-execute" can be repeated many times
- It is also possible that a significant part of the code may be executed successfully before the interpreter finds an error.
- All Pythons coming from the PSF are written in the "C" language.

2 Basics

2.1 Notes

- Python requires that there cannot be more than one instruction in a line.
- Numbers are converted into machine representation (a set of bits)
- **exponentiation (power) operator uses right-sided binding**
- Python treats the sign = not as equal to, but as assign a value
- Python has **left-sided binding**
- Any entity recognizable by Python can play the role of a function argument, although it has to be assured that the function is able to cope with it.
- Any entity recognizable by Python can be a function result.
- although using a Greek letter to name a variable is fully possible in Python, the symbol of pi number is named `pi` – it's a more convenient solution, especially for that part of the world which neither has nor is going to use a Greek keyboard
- Python, although very powerful,*isn't omnipotent* – it's forced to use many helpers if it's going to process files or communicate with physical devices;

2.2 Function arguments

Python functions may accept any number of arguments, as many as necessary to perform their tasks, any number includes zero.

2.3 Function invocation

The function name along with the parentheses and **argument(s)**, forms the function invocation.

2.4 Positional way

this name comes from the fact that the meaning of the argument is dictated by its position, the second argument will be outputted after the first, not the other way round.

2.5 Keyword arguments

The name stems from the fact that the meaning of these arguments is taken not from its location (position) but from the special word (keyword) used to identify them.

- a keyword argument consists of three elements:
  - a keyword identifying the argument
  - an equal sign (=)
  - and a value assigned to that argument;
  - any keyword arguments have to be put **after the last positional argument** (this is very important)

2.6 Print keyword arguments

- end
- sep

2.7 Literal

A literal is data whose values are determined by the literal itself.

2.8 Octal Representation

If an integer number is preceded by an 0O or 0o prefix (zero-o), it will be treated as an octal value.

2.9 Hexadecimal Representation

Such numbers should be preceded by the prefix 0x or 0X (zero-x).

2.10 Scientific Notation

- 3E8 (3 per 10 power 8)
- 6E-38 (6 per 10 power -38)
- 2e-10 (0.0000000002)

2.11 Operation

An operator is a symbol of the programming language, which is able to operate on the values.

2.12 Expressions

Data and operators when connected together form expressions.

2.13 Power

A ** (double asterisk) sign is an exponentiation (power) operator.

2.14 Overloading

The ability to use the same operator against completely different kinds of data (like numbers vs. strings) is called overloading (as such an operator is overloaded with different duties).

2.15 integer divisional operator

A // (double slash) sign is an integer divisional operator. It differs from the standard / operator in two details:

- its result lacks the fractional part – it's absent (for integers), or is always equal to zero (for floats); this means that the results are always rounded;
- **This is very important – rounding always goes to the lesser integer**
  - 6 // 4 ==> 1
  - -6 // 4 ==> -2 (-2 is lesser than -1)
  - 6 // -4 ==> -2 (-2 is lesser than -1)
- 12 % 4.5 ==> 3.0 beacause:
  1. 12 // 4.5 ==> 2.0
  2. 2.0 * 4.5 ==> 9.0
  3. 12 - 12.0 ==> 3.0

2.16 hierarchy of priorities

The phenomenon that causes some operators to act before others is known as the hierarchy of priorities.

2.17 Variable's naming

- the name of the variable must be composed of upper-case or lower-case letters, digits, and the character _ (underscore)
- the name of the variable must begin with a letter
- the underscore character is a letter
- upper- and lower-case letters are treated as different
- the name of the variable must not be any of Python's reserved words
- Moreover, Python lets you use not only Latin letters but also characters specific to languages that use other alphabets.

2.18 How python treats comments

Whenever Python encounters a comment in your program, the comment is completely transparent to it – from Python's point of view, this is only one space

2.19 int() and float fails

- If int() or float() functions fail to convert the string to integer, the whole program will fail too
- int() function **does not** round the number

2.20 Plus sing

- It can be used more than a time in an expression and in such a context

it behaves according to left-sided binding.

- It is a binary operator with left-sided binding

2.21 Cascade

The way to assemble subsequent if-elif-else statements is sometimes called a **cascade**.

2.22 pseudocode

we'll use a kind of notation which is not an actual programming language (it can be neither compiled nor executed), but it is formalized, concise and readable.

2.23 Equivalent conditions:

1. if n % 2 == 1 *is equal to* if n % 2
2. while n % 2 != 0 *is equal to* while n % 2

- these are not have to be just 0/1, but anything except 0 is considered as True
- Logical operators take their arguments as a whole regardless of how many bits they contain. The operators are aware only of the value: zero (when all the bits are reset) means False; not zero (when at least one bit is set) means True.

2.24 Loop else branch

The loop's else branch is always executed once, **regardless** of whether the loop has entered its body or not.

2.25 logical operator

conjunction and

disjunction or

2.26 bitwise operators

However, there are four operators that allow you to manipulate single bits of data. They are called bitwise operators g

1. & (ampersand) bitwise conjunction
2. | (bar) bitwise disjunction
3. ~ (tilde) bitwise negation
4. ^ (caret) bitwise exclusive or (xor)

2.26.1 To make it easy:

- & requires exactly two 1s to provide 1 as the result
- | requires at least one 1 to provide 1 as the result
- ^ requires exactly one 1 to provide 1 as the result

2.27 Bit mask ::

to Determine if FlagRegister & myMask: #is set

to Reset FlagRegister &= ~myMask

to Set FlagRegister |= myMask

to negate FlagRegister ^= myMask

2.28 Shifting

- Shifting is applied only to integer values
- as two is the base for binary numbers (not 10), shifting a value one bit to the left thus corresponds to multiplying it by two; respectively, shifting one bit to the right is like dividing by two (notice that the rightmost bit is lost). The shift operators in Python are a pair of diagraphs, « and », clearly suggesting in which direction the shift will act. The left argument of these operators is an integer value whose bits are shifted. The right argument determines the size of the shift. It shows that this operation is certainly not commutative.

2.29 List Elements

You can't access an element which doesn't exist – you can neither get its value nor assign it a value, so you need append() method

2.30 Method ~ Function

A method is owned by the data it works for, while a function is owned by the whole code.

2.31 list.insert(where, what)

2.32 to change value of 2 variables

var1, var2 = var2, var1

2.32.1 also for lists exp:

num[0], num[4] = num[4], num[0]

2.33 list ~ variables

- the name of an ordinary variable is the name of its content
- the name of a list is the name of a memory location where the list is stored. so `list2 = list1` copies the name of the array, not its contents. In effect, the two names (list1 and list2) identify the same location in the computer memory. Modifying one of them affects the other, and vice versa.

2.34 slice

syntax `list[start:end]` this contains from start to end-1

slice is an element of Python syntax that allows you to make a brand new copy of a list, or parts of a list. It actually copies the list's contents, not the list's name

`list2 = list1[:]` copy content of the list1 to the list2

exp `list1 = [1, 2, 5, 4, 8, 0] list2 = list1[1:3]` this will result the list2 as [2,5] which contain first and second element of list1 but not third one

2.35 del

syntax `del list[start:end]`

exp `del list[:]` will empty the list

syntax `del list` will delete the list

2.36 in ~ not in

- `in` checks if a given element (its left argument) is currently stored somewhere inside the list

- `not in` checks if a given element (its left argument) is absent in a list

2.37 list in list

- `board[[ i for i in range(8)] for j in range(8)]` this create a 8X8 matrix

2.38 function invocation

- As python is a scripting language You mustn't invoke a function which is not known at the moment of invocation.
- You mustn't have a function and a variable of the same name. Assigning a value to the a name of a function causes Python to forget its previous role and the function will become unavailable.

2.39 function's Parameter

- A parameter is actually a variable, but there are two important factors that make parameters different and special:
  1. they exist only inside functions in which they have been defined, and the only place where the parameter can be defined is a space between a pair of parentheses in the def statement
  2. assigning a value to the parameter is done at the time of the function's invocation, by specifying the corresponding argument.

2.40 Parameter ~ Argument

- A parameter is a variable in a method definition. When a method is called, the arguments are the data you pass into the method's parameters. In fact Parameter is variable in the declaration of function. Argument is the actual value of this variable that gets passed to function.
- **important**, specifying one or more parameters in a function's definition is also a requirement, and **you have** to fulfill it during invocation.

You must provide as many arguments as there are defined parameters. Failure to do so will cause an error.

2.41 Shadowing

variable shadowing occurs when a variable declared within a certain scope (like function) has the same name as a variable declared in an outer scope.

2.42 Function's data passing

Positional parameter passing A technique which assigns the i'th (first, second, and so on) argument to the i'th (first, second, and so on) function parameter is called *positional parameter passing*

Positional arguments while arguments passed in the above way are named *positional arguments.*

keyword argument passing Python offers another convention for passing arguments, where the meaning of the argument is dictated by its name, not by its position – it's called keyword argument passing.

Note that You can mix both fashions if you want – there is only one unbreakable rule: you have to put positional arguments before keyword ones.

2.43 return

just `return` it causes the immediate termination of the function's execution, and an instant return (hence the name) to the point of invocation. if a function is not intended to produce a result, using the return instruction is not obligatory – it will be executed implicitly at the end of the function.

- Don't forget this: if a function doesn't return a certain value using a return expression clause, it is assumed that it implicitly returns None.

`return expression` it causes the immediate termination of the function's execution and moreover, the function will evaluate the expression's

value and will return (hence the name once again) it as the function's result.

2.44 None

- data of this value doesn't represent any reasonable value – actually, it's not a value at all; hence, it mustn't take part in any expressions.
- There only two kinds of circumstances when None can be safely used:
  1. when you assign it to a variable (or return it as a function's result)
  2. when you compare it with a variable to diagnose its internal state.

2.45 scope (very important)

2.45.1 variables

- The scope of a name (e.g., a variable name) is the part of a code where the name is properly recognizable.
- a variable existing outside a function has a scope inside the functions' bodies. But if you want to modify it you should use **global** key word inside the function to make changes globaly, if not you will define an internal variable inside the function
- so a variable existing outside a function has a scope inside the function's bodies, excluding those of them which define a variable of the same name.
- It also means that the scope of a variable existing outside a function is supported only when getting its value
- the variable created inside the function is not the same as when defined outside it – it seems that there two different variables of the same name;
- moreover, the function's variable shadows the variable coming from the outside world.

2.45.2 list

- for lists it's totally different as demonstrated below

- in any condition you can modify list using method inside or outside function, supriselly even if you modify list which is passed as argument to function with it's parameter name, this will modify global list
- But if you modify without methods via parameter name it will modify just inside function

2.45.3 conclusion

2.45.3.1 with parameter

action	effect
if assign something via parameter name	just effects local paramet
if modify using parameter name via methods	effects globaly and localy
if modify using list name via methods	effects globaly and localy

2.45.3.2 without parameter

action	effect
if assign something via global list name	creates a new list localy and does not effect global list
if modify using list name via methods	effects globaly
if modify using non existence parameter via methods	run-time error

2.45.4 Sample Code

```
# in this sectoin i'm going to examin scope for list

list1 = [1, 2, 3, 4, 5]
list2 = [1, 2, 3, 4, 5]
list3 = [1, 2, 3, 4, 5]
list4 = [1, 2, 3, 4, 5]
list5 = [1, 2, 3, 4, 5]


# this will test just modification inside the function
def test1(lst):
```

```
12      print("Test 1 lst", lst)
13      lst = [1, 2]
14      print('Test 1 modified lst:  ', lst)
15      print("Test 1 list:", list1)
16
17
18  print("----------TEST 1----------")
19  test1(list1)
20  print("outside", list1)
21
22
23  # now lets test list modification via methods inside the function
24  def test2(lst):
25      print("Test 2 lst", lst)
26      lst.append(6)
27      print("Test 2 modified lst:  ", lst)
28      print("Test 2 list:", list2)
29
30
31  print("----------TEST 2----------")
32  print("outside before function invocation:", list2)
33  test2(list2)
34  print("outside:", list2)
35
36
37  # now lets test list modification via methods outside the function
38  def test3(lst):
39      print("Test 3 lst", lst)
40      list3.append(6)
41      print("Test 3 lst after modifing list3 :  ", lst)
42      print("Test 3 list:", list3)
43
44
45  print("----------TEST 3----------")
46  print("outside before function invocation:", list3)
47  test3(list3)
48  print("outside:", list3)
49
50
51  # Test list modification without passing as argument via methods
```

```
52 def test4():
53     list4.append(6)
54     print("Test 4 list:", list4)
55 
56 
57 print("----------TEST 4----------")
58 print("outside before function invocation:", list4)
59 test4()
60 print("outside:", list4)
61 
62 
63 
64 # Test list modification without passing as argument without methods
65 def test5():
66     list5 = [1, 2]
67     print("Test 5 inside function list5 :", list5)
68 
69 
70 print("----------TEST 5----------")
71 print("outside before function invocation list5:", list5)
72 test5()
73 print("outside after invocation list5:", list5)
```

Code Result

```
----------TEST 1----------
Test 1 lst [1, 2, 3, 4, 5]
Test 1 modified lst:   [1, 2]
Test 1 list: [1, 2, 3, 4, 5]
outside [1, 2, 3, 4, 5]
----------TEST 2----------
outside before function invocation: [1, 2, 3, 4, 5]
Test 2 lst [1, 2, 3, 4, 5]
Test 2 modified lst:   [1, 2, 3, 4, 5, 6]
Test 2 list: [1, 2, 3, 4, 5, 6]
outside: [1, 2, 3, 4, 5, 6]
----------TEST 3----------
outside before function invocation: [1, 2, 3, 4, 5]
Test 3 lst [1, 2, 3, 4, 5]
Test 3 lst after modifing list3 :   [1, 2, 3, 4, 5, 6]
Test 3 list: [1, 2, 3, 4, 5, 6]
```

```
outside: [1, 2, 3, 4, 5, 6]
----------TEST 4----------
outside before function invocation: [1, 2, 3, 4, 5]
Test 4 list: [1, 2, 3, 4, 5, 6]
outside: [1, 2, 3, 4, 5, 6]
----------TEST 5----------
outside before function invocation list5: [1, 2, 3, 4, 5]
Test 5 inside function list5 : [1, 2]
outside after invocation list5: [1, 2, 3, 4, 5]
```

2.46 global

- Using this keyword inside a function with the name (or names separated with commas) of a variable(s), forces Python to refrain from creating a new variable inside the function – the one accessible from outside will be used instead. In other words, this name becomes global (it has a global scope, and it doesn't matter whether it's the subject of read or assign).
- if the argument is a list, then changing the value of the corresponding parameter doesn't affect the list
- but if you change a list identified by the parameter (note: the list, not the parameter!), the list will reflect the change.

2.47 recursion

recursion is a technique where a function invokes itself.

2.48 sequence type

A sequence type is a type of data in Python which is able to store more than one value (or less than one, as a sequence may be empty), and these values can be sequentially (hence the name) browsed, element by element. As the for loop is a tool especially designed to iterate through sequences, we can express the definition as: **a sequence is data which can be scanned by the for loop.** like **list**

2.49 mutability

It is a property of any of Python's data that describes its readiness to be freely changed during program execution. There are two kinds of Python data: mutable and immutable.

2.49.1 mutable

Mutable data can be freely updated at any time – we call such an operation in situ. In situ is a Latin phrase that translates as literally "in position". For example, the following instruction modifies the data in situ: `list.append(1)`

2.49.2 immutable

Immutable data cannot be modified in this way. Imagine that a list can only be assigned and read over. You would be able neither to append an element to it, nor remove any element from it. This means that appending an element to the end of the list would require the recreation of the list from scratch. You would have to build a completely new list, consisting of the all elements of the already existing list, plus the new element.

2.50 tuple

A tuple is an **immutable sequence type**. It can behave like a list, but it mustn't be modified in situ.

2.50.1 syntax

name = (value, value, value) or name = value, value, value

- value could be anything hence float, int, str, ...
- **If you want to create a one-element tuple, you have to take into consideration the fact that, due to syntax reasons**

2.51 dictionary

The dictionary is another Python data structure. It's not a sequence type (but can be easily adapted to sequence processing) and it is mutable.

- each key must be unique – it's not possible to have more than one key of the same value;
- a key may be data of any type: it may be a number (integer or float), or even a string;
- a dictionary is not a list – a list contains a set of numbered values, while a dictionary holds pairs of values;
- the len() function works for dictionaries, too – it returns the numbers of key–value elements in the dictionary
- a dictionary is a one-way tool – if you have an English–French dictionary, you can look for French equivalents of English terms, but not vice versa.
- The order in which a dictionary stores its data is completely out of your control, and your expectations. That's normal.

2.51.1 syntax

name = { key:value, key:value }

2.51.2 access

- name[key]
- via keys() method

```
for key in dct.keys():
    print(dct[key])
```

- via values() method

```
for value in dct.values():
    print(values)
```

- via items() method which return a list of tuples

```
for key, value in dct.items():
    print(key, value)
```

2.51.3 add

name[newKey] = newValue **180 degree different from lists**

2.51.4 del

del name[key]

2.52 Table of priorities

! ~ (type) ++ -- + -	unary
* / %	
+ -	binary
<< >>	
< <= > >=	
== !=	
&	
|	
&&	
||	
= += -= *= /= %= &= ^= |= >>= <<=	

Figure 1: Priority Table

3 Modules and Packages

3.1 Namespace

Namespace is a space in which some names exist and the names don't conflict with each other; in fact **there are not** two different objects of the same name.

3.2 Module

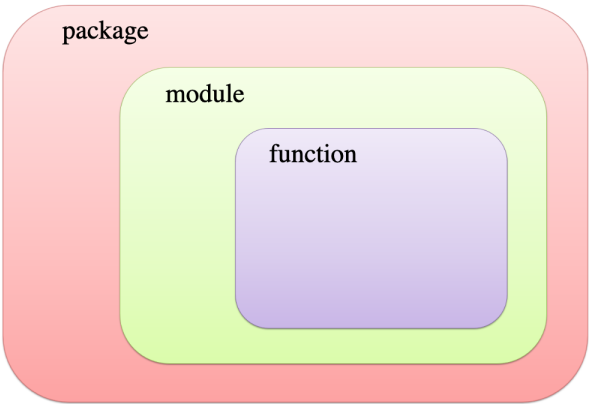

Figure 2: Modules and Packages Hierarchy

Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module; a module is a **kind of container** filled with functions. You can pack as many functions as you want into one module and distribute it across the world;

- First of all, a module is identified by its name.
- All modules, along with the built-in functions, form the "Python standard library". Python Standard Library
- Each module consists of entities (like a book consists of chapters). These entities can be functions, variables, constants, classes, and objects. If you know how to access a particular module, you can make use of any of the entities it stores.

3.3 Package

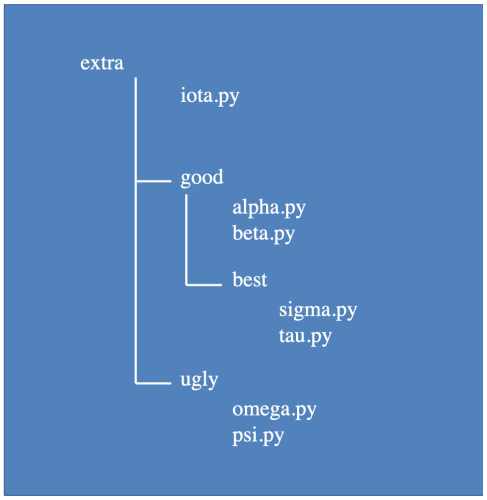

Figure 3: Package example tree

Making many modules may cause a little mess – sooner or later you'll want to group your modules exactly in the same way as you've previously grouped functions, the solution is a **package**; in the world of modules, a package plays a similar role to a folder/directory in the world of files.

3.3.1 Locating package's files

for instance according to the image above:

- the location of a function named FunT() from the tau package may be described as:

  `extra.good.best.tau.FunT()`

- a function marked as below comes from the `psi` module being stored in the `ugly` subpackage of the `extra` package.

  `extra.ugly.psi.FunP()`

3.3.2 Initialization package

The initialization of a module is done by an unbound code (not a part of any function) located inside the module's file. As a package is not a file, this technique is useless for initializing packages. You need to use a different trick instead – Python expects that there is a file with a very unique name inside the package's folder `__init__.py`. The content of the file is executed when any of the package's modules is **imported**. If you don't want any special initializations, you can leave the file empty, but you mustn't omit it.

- Note: it's not only the "root" folder that can contain the `__init__.py` file – you can put it inside any of its subfolders (subpackages) too. It may be useful if some of the subpackages require individual treatment and special kinds of initialization.

3.4 Import

syntax `import moduleName1, moduleName2, ...`

- The instruction may be located anywhere in your code, but it must be placed before the first use of any of the module's entities.
- The instruction imports two modules, first the one named `moduleName1` and then the second named `moduleName2`.

syntax `import moduleName as alias`

- Aliasing causes the module to be identified under a different name than the original. This may shorten the qualified names, too.
- after successful execution of an aliased import, the original module name becomes **inaccessible** and must not be used.

If the module of a specified name exists and is accessible (a module is in fact a Python source file), Python imports its contents, i.e., all the names defined in the module become known, but they **don't enter your code's namespace.** So to use something from imported module use have to specify it's module name to avoid conflicts between your namespace and module namespace. Simply put:

- The name of the module
- a dot;

- The name of the entity

syntax `from moduleName import entity`

- the listed entities (and only those ones) are imported from the indicated module;
- the names of the imported entities **are accessible without qualification**.
- In this method if you assign anything to any of imported entities or define a function with their name, you will shadow the module's entities and from now on you don't have access to module entities anymore!
- **Vise versa,** if you import module's entities after some value or functions which has the same name, the module entities will shadow them.

syntax `from moduleName import *`

- This is like the previous condition but import all module's entities
- **Be careful**, in this way if you don't know some module's entities, you may cause a conflict or name shadowing

syntax `from moduleName import entity as alias, entity as alias, entity as alias, ...`

- In turn, when you use the `from module import name` variant and you need to change the entity's name, you make an alias for the entity. This will cause the name to be replaced by the alias you choose.
- As previously, the original (unaliased) name becomes inaccessible.

Note: import is also a keyword (with all the consequences of this fact).

3.4.1 import addressing

- If the module is not in the same directory but the directory is a child of current directory:
  - `import directoryName.moduleName`
  - OR

```
from sys import path
path.append("./directoryName")
import moduleName
```

- If the module is not in the same directory and the directory is not a child of current directory:
  - To be like:

```
from sys import path
path.append("path/to/module's/directory")
import moduleName
```

- Let assume we have a package named **extra**, if we zip hole files and directories in a zip file named **extrapack.zip** now we can **append** this file to the **path** variable and by then we can treat the package as it's name is **extra** not **extrapack.zip**!!! for instance `import extra.good.best.sigma as sig` . Because python treats zip files almost as regular file/directories
- Remember if you import the hole module or package, every time you want to use them you have to specify **fully qualified path** to them, if you don't like it:
  - use `from ... import ...`
  - or use alias `import ... as ...`
- We've used the append() method – in effect, the new path will occupy the last element in the path list; if you don't like the idea, you can use insert() instead.
- The above method works as the same also for packages

3.5 Creating a Module

1. Creating a file with module name and .py extension
2. Creating a file which is named **main.py**, it contains just a line like `import module` (of course it's not part of creating a module, we made it to test importing the module)
3. Having this two files and executing main.py, a directory will be created which is named `__pycache__`. This directory contains a file most like **module.cpython-xy.pyc**
   - The name of the file is the same as your module's name
   - he part after the first dot says which Python implementation has created the file (CPython here) and its version number. (xy)
   - The last part (pyc) comes from the words "Python" and "compiled"

3.5.1 Notes:

- When Python imports a module for the first time, it translates its contents into a somewhat compiled shape. The file doesn't contain machine code – it's internal Python semi-compiled code, ready to be executed by Python's interpreter. As such a file doesn't require lots of the checks needed for a pure source file, the execution starts faster, and runs faster, too. Python is able to check if the module's source file has been modified (in this case, the **pyc** file will be rebuilt) or not (when the **pyc** file may be run at once). As this process is fully automatic and transparent, you don't have to keep it in mind.
- When a module is imported, its content is **implicitly executed** by Python. Be careful, so for example if you have a print() barely in you module, it will execute!. But in fact it gives the module the chance to initialize some of its internal aspects (e.g., it may assign some variables with useful values). The initialization takes place **only once**, when the first import occurs, so the assignments done by the module aren't repeated unnecessarily.
  - Imagine the following context:
    - there is a module named `mod1`;

* there is a module named `mod2` which contains the import `mod1` instruction;
* there is a main file containing the import `mod1` and import `mod2` instructions.

At first glance, you may think that mod1 will be imported twice fortunately, **only the first import occurs.** Python remembers the imported modules and silently omits all subsequent imports.

3.5.2 `__name__` variable

- when you run a file directly, its `__name__` variable is set to `__main__`;
- when a file is imported as a module, its `__name__` variable is set to the file's name (excluding .py)

3.5.3 variable deceleration

Unlike many others programming languages, Python has no means of allowing you to hide such variables from the eyes of the module's users. You can only inform your users that this is your variable, that they may read it, but that they should not modify it under any circumstances. This is done by preceding the variable's name with `_` or `__`, but remember, it's only a convention. Your module's users may obey it or they may not. Exp, `__counter = 0`

3.5.4 shabang

syntax `#!/usr/bin/env python3`

For Unix and Unix-like OSs (including MacOS) such a line instructs the OS how to execute the contents of the file (in other words, what program needs to be launched to interpret the text). In some environments (especially those connected with web servers) the absence of that line will cause trouble;

3.5.5 doc-string

syntax `"""the module description"""`

a string (maybe a multiline) placed before any module instructions (including imports) is called the doc-string, and should briefly explain the purpose and contents of the module;

3.6 path

There's a special variable (actually a list) storing all locations (folders/directories) that are searched in order to find a module which has been requested by the import instruction. Python browses these folders in the order in which they are listed in the list – if the module cannot be found in any of these directories, the import **fails**. Otherwise, **the first folder** containing a module with the desired name will be taken into consideration and *if any of the remaining folders contains a module of that name it will be* ***ignored***. The variable is named `path`, and it's accessible through the module named `sys`.

- there is a zip file listed as one of the path's elements – it's not an error. Python is able to treat zip files as ordinary folders – this can save lots of storage.
- the folder in which the execution starts is listed in the first path's element.

3.7 dir()

it is able to reveal all the names provided through a particular module. There is one condition: the module has to have been previously imported as a whole (i.e., using the import module instruction – from module is not enough). The function returns an alphabetically sorted list containing all entities' names available in the module identified by a name passed to the function as an argument.

Note: if the module's name has been aliased, you must use the alias, not the original name.

3.8 Some math functions

3.8.1 pow()

pow(x,y) This is a built-in function, and doesn't have to be imported.

3.8.2 floor() & ceil()

floor always round to smaller number, where ceil round to bigger number

```
from math import ceil, floor, trunc

x = 1.4
y = 2.6
print(floor(x), floor(y))
print(floor(-x), floor(-y))
print(ceil(x), ceil(y))
print(ceil(-x), ceil(-y))
print(trunc(x), trunc(y))
print(trunc(-x), trunc(-y))
```

Code Result

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

3.8.3 random()

- Produces a float number x coming from the range (0.0, 1.0) –in other words: (0.0 <= x < 1.0).
- A random number generator takes a value called a seed, treats it as an input value, calculates a "random" number based on it.

3.8.4 seed()

- `seed()` – sets the seed with the current time;
- `seed(i)` – sets the seed with the integer value i.

3.8.5 randrange()

If you want integer random values, one of the following functions would fit better. First three one has **right-sided exclusion!** but the last one `randint(left, right)` starts from *left* and ends on *right* which also could contain *right* integer

1. randrange(end)
2. randrange(beg,end)
3. randrange(beg, end, step)
4. randint(left, right)

```
from random import randrange, randint

print(randrange(5))
print(randrange(0,5))
print(randrange(0,20,2))
print(randint(0,1))
```

Code Result

```
1
0
8
0
```

3.8.6 choice()

syntax `choice(sequence)`

Chooses a "random" element from the input sequence(list,...) and returns it.

3.8.7 sample()

syntax `sample(sequence, elements_to_chose=1)`

Chooses some of the input elements(default in one), returning a list with the choice. The elements in the sample are placed in random order. Note:

the `elements_to_chose` must not be greater than the length of the input sequence.

```
from random import sample
lst = list(range(0, 10))
print(sample(lst, 5))
```

Code Result

```
[9, 5, 2, 7, 1]
```

3.9 Some platform functions

3.9.1 platform()

syntax

```
from platform import platform
platform(aliased=False, terse=False)
```

- aliased→ when set to True (or any non-zero value) it may cause the function to present the alternative underlying layer names instead of the common ones;
- terse→ when set to True (or any non-zero value) it may convince the function to present a briefer form of the result (if possible)

3.9.2 machine()

syntax

```
from platform import machine
print(machine())
```

Code Result

```
x86_64
```

- Sometimes, you may just want to know the generic name of the processor which runs your OS together with Python and your code – a function named machine() will tell you that. As previously, the function returns a string.

3.9.3 processor()

syntax

```
from platform import processor
print(processor())
```

Code Result

```
i386
```

- The processor() function returns a string filled with the real processor name (if possible)

3.9.4 system()

syntax

```
from platform import system
print(system())
```

Code Result

```
Darwin
```

- A function named system() returns the generic OS name as a string

3.9.5 version()

syntax

```
from platform import version
print(version())
```

Code Result

```
Darwin Kernel Version 18.2.0: Mon Nov 12 20:24:46 PST 2018; root:xnu-4903.231.4~2/REI
```

- The OS version is provided as a string by the version() function

3.9.6 python info

syntax

```
from platform import python_implementation, python_version_tuple
print(python_implementation())
print(python_version_tuple())
```

Code Result

```
CPython
('3', '7', '1')
```

- `python_implementation()` → returns a string denoting the Python implementation (expect 'CPython' here, unless you decide to use any non-canonical Python branch)
- `python_version_tuple()` → returns a three-element tuple filled with:
  - the major part of Python's version;
  - the minor part;
  - the patch level number
  - for exp: ('3', '7', '0')

4 Error and Exception

4.1 Raising an exception

Each time your code tries to do something wrong/foolish/irresponsible/crazy/unenforceable, Python does two things:

- it stops your program;
- it creates a special kind of data, called an exception.

Both of these activities are called raising an exception.We can say that Python always raises an exception (or that an exception has been raised) when it has no idea what do to with your code.

4.2 try - except

1. the `try` keyword begins a block of the code which may or may not be performing correctly;
2. next, Python tries to perform the risky action; if it fails, an exception is raised and Python starts to look for a solution;
3. `except` keyword starts a piece of code which will be executed if anything inside the try block goes wrong – if an exception is raised inside a previous try block, it will fail here, so the code located after the except keyword should provide an adequate reaction to the raised exception;
4. **returning** to the previous nesting level ends the try-except section.

Sample code:

```
a = 1
b = 0

try:
    print("START")
    print(a / b)
except:
    print('It cannot be done!')

print('THE END')
```

Code Result

```
START
It cannot be done!
THE END
```

Note: Code block inside the `try` will execute until first error and after that python will immediately jumps out of the block and into the first instruction located after the except: keyword; this means that **some** of the instructions from the block may be silently omitted.

4.2.1 try - multiple exceptions

- syntax
  - if an unnamed except branch exists (one without an exception name), it has to be specified as the last.
  - In case you do not have any general exception(unnamed except), if none of the specified named except branches matches the raised exception, the exception remains unhandled.

```
try:
    .
    .
except exc1:
    .
    .
except exc2:
```

```
    .
    .
except:
    .
    .
```

```
try:
    .
    .
except (exc1, exc2)
    .
    .
except:
    .
    .
```

- Sample code

```
try:
    x = 0
    y = 1 / x
    print(y)
except ZeroDivisionError:
    print('Cannot divide by zero sorry')
except ValueError:
    print('You have to enter an integer value')
except:
    print('Oh, dear')

print('THE END')
```

Code Result

```
Cannot divide by zero sorry
THE END
```

4.2.2 finally

Code in a finally statement even runs if an uncaught exception occurs in one of the preceding blocks.

```
try:
    print(1)
    print(10 / 0)
except ZeroDivisionError:
    print('unknown_var')
finally:
    print("This is executed last")
```

Code Result

```
1
unknown_var
This is executed last
```

4.3 Anatomy of exceptions

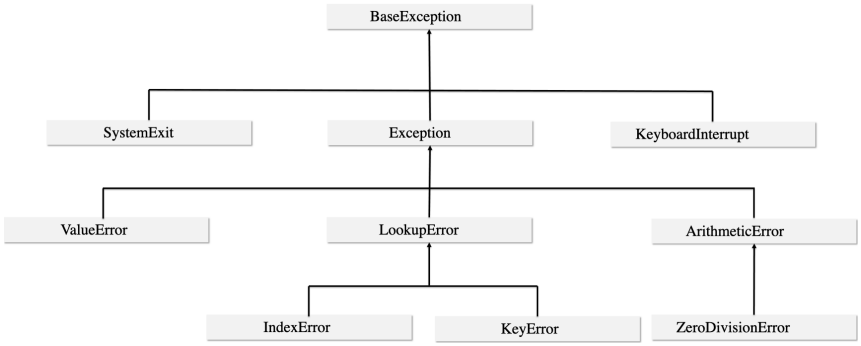

Figure 4: Exception tree

Some of the built-in exceptions are more general (they include other exceptions) while others are completely concrete (they represent themselves only). We can say that the closer to the root an exception is located, the more general (abstract) it is. In turn, the exceptions located at the branches' ends (we can call them leaves) are concrete.

- **The order of the branches matters!**

- don't put more general exceptions before more concrete ones;
- each raised exception falls into the first matching branch;
- the matching branch doesn't have to specify the same exception exactly – it's enough that the exception is more general (more abstract) than the raised one.

4.4 Some useful exceptions

4.4.1 BaseException ← Exception ← ArithmeticError

an abstract exception including all exceptions caused by arithmetic operations like zero division or an argument's invalid domain

4.4.2 BaseException ← Exception ← AssertionError

a concrete exception raised by the assert instruction when its argument evaluates to **False, None, zero, or an empty string**

```
1 from math import tan,radians
2 angle = int(input('Enter integral angle in degrees: '))
3 # we must be sure that angle != 90 + k*180
4 assert angle % 180 != 90
5 print(tan(radians(angle)))
```

4.4.3 BaseException

the most general (abstract) of all Python exceptions – all other exceptions are included in this one; it can be said that the following two except branches are equivalent:

```
except:
```

```
except BaseException:
```

4.4.4 BaseException ← Exception ← LookupError ← IndexError

a concrete exception raised when you try to access a non-existent sequence's element (e.g., a list's)

```
# the code shows an extravagant way of leaving the loop
list = [1,2,3,4,5]
ix = 0
doit = True
while doit:
    try:
        print(list[ix])
        ix += 1
    except IndexError:
        doit = False
print('Done')
```

4.4.5 BaseException ← Exception ← LookupError

an abstract exception including all exceptions caused by errors resulting from invalid references to different collections (lists, dictionaries, tuples, etc.)

4.4.6 BaseException ← KeyboardInterrupt

a concrete exception raised when the user uses a keyboard shortcut designed to terminate a program's execution (Ctrl-C in most OSs); if handling this exception doesn't lead to program termination, the program continues its execution. Note: this exception is not derived from the Exception class.

```
# this code cannot be terminated by pressing Ctrl-C
from time import sleep
seconds = 0
while True:
    try:
        print(seconds)
        seconds += 1
        sleep(1)
    except KeyboardInterrupt:
        print("Don't do that!")
```

4.4.7 BaseException ← Exception ← MemoryError

a concrete exception raised when an operation cannot be completed due to a lack of free memory

```
# this code causes the MemoryError exception
# warning: executing this code may be crucial for your OS
# don't run it in production environments!

string = 'x'
try:
    while True:
        string = string + string
        print(len(string))
except MemoryError:
    print('This is not funny!')
```

4.4.8 BaseException ← Exception ← ArithmeticError ← OverflowError

a concrete exception raised when an operation produces a number too big to be successfully stored

```
# the code prints subsequent values of exp(k), k = 1,2,4,8,16,...
from math import exp
ex = 1
try:
    while True:
        print(exp(ex))
        ex *= 2
except OverflowError:
    print('Number is too big.')
```

4.4.9 BaseException ← Exception ← StandardError ← ImportError

a concrete exception raised when an import operation fails

```
# one of this imports will fail - which one?
try:
```

```
    import math
    import time
    import abracadabra
except:
    print('One of your imports has failed. ')
```

4.4.10 BaseException ← Exception ← LookupError ← KeyError

a concrete exception raised when you try to access a non-existent collection's element (e.g., a dictionary's)

```
# how to abuse the dictionary and how to deal with it
dict = { 'a' : 'b', 'b' : 'c', 'c' : 'd' }
ch = 'a'
try:
    while True:
        ch = dict[ch]
        print(ch)
except KeyError:
    print('No such key:', ch)
```

Code Result

```
b
c
d
No such key: d
```

4.5 raise

The raise instruction raises the specified exception named exc as if it was raised in a normal

- **simulate** raising actual exceptions (e.g., to test your handling strategy)
- **partially handle** an exception and make another part of the code responsible for completing the handling (separation of concerns).
- The `raise` instruction may also be utilized without any exception name.

- this kind of raise instruction may be used inside the except branch **only**;
- using it in any other context **causes an error**.
- The instruction will **immediately re-raise the same exception as currently handled.**

```
def badfun(n):
    try:
        return n/0
    except:
        print('I did it again!')
        raise

try:
    badfun(0)
except ArithmeticError:
    print('I see!')
print('THE END')
```

Code Result

```
I did it again!
I see!
THE END
```

- Exceptions can be raised with arguments that give detail about them.

```
name = "123"
raise NameError("Invalid name!")
```

4.6 assert

How does it work?

- It evaluates the expression;
- if the expression evaluates to True, or a non-zero numerical value, or a non-empty string, or any other value different than None and False, it won't do anything else;
- otherwise, it automatically and immediately raises an exception named AssertionError (in this case, we say that the assertion has failed)

How it can be used?

- you may want to put it into your code where you want to be absolutely safe from evidently wrong data, and where you aren't absolutely sure that the data has been carefully examined before (e.g., inside a function used by someone else)
- raising an `AssertionError` exception secures your code from producing invalid results, and clearly shows the nature of the failure;
- assertions don't supersede exceptions or validate the data – they are their supplements.
- It's also very good to use it for documentation of your code

```
import math
x = float(input())
assert x>=0.0
x = math.sqrt(x)
print(x)
```

- The assert can take a second argument that is passed to the AssertionError raised if the assertion fails.

```
temp = -10
assert (temp >= 0), "Colder than absolute zero!"
```

4.7 else

A code labelled in this way is executed when (and only when) no exception has been raised inside the `try` part. We can say that exactly one branch can be executed after `try`: either the one beginning with `except` (don't forget that there can be more than one branch of this kind) or the one starting with `else`. Note that the else branch has to be located after the last except branch.

```
def reciprocal(n):
        try:
                n = 1 / n
        except ZeroDivisionError:
                print("Division failed")
                return None
        else:
```

```
8                   print("Everything went fine")
9                   return n
10
11
12  print(reciprocal(2))
13  print(reciprocal(0))
```

4.8 finally

The `try-except` block can be extended in one more way – by adding a part headed by the `finally` keyword (it must be the last branch of the code designed to handle exceptions). Note that these two variants (`else` and `finally`) aren't dependent in any way, and they can coexist or occur independently. The finally block is always executed (it finalizes the try-except block execution, hence its name), no matter what happened earlier, even when raising an exception, no matter whether this has been handled or not.

```
1   def reciprocal(n):
2           try:
3                   n = 1 / n
4           except ZeroDivisionError:
5                   print("Division failed")
6                   n = None
7           else:
8                   print("Everything went fine")
9           finally:
10                  print("It's time to say good bye")
11                  return n
12
13  print(reciprocal(2))
14  print(reciprocal(0))
```

4.9 Exception Object

You probably won't be surprised to learn that exceptions are classes. Furthermore, when an exception is raised, an object of the class is instantiated, and goes through all levels of program execution, looking for the except branch that is prepared to deal with it. Such an object carries some useful

information which can help you to precisely identify all aspects of the pending situation. To achieve that goal, Python offers a special variant of the exception clause

In the code below you can see, the except statement is extended, and contains an additional phrase starting with the as keyword, followed by an identifier. The identifier is designed to catch the exception object so you can analyze its nature and draw proper conclusions. Note that the identifier's scope covers its `except` branch, and doesn't go any further.

```
try:
        i = int("hello!")
except Exception as e:
        print(e)
        print(e.__str__())
```

4.10 Extending Exceptions

The exceptions hierarchy is neither closed nor finished, and you can always extend it if you want or need to create your own world populated with your own exceptions. It may be useful when you create a complex module which detects errors and raises exceptions, and you want the exceptions to be easily distinguishable from any others brought by Python.

This is done by defining your own, new exceptions as subclasses derived from predefined ones. Note that if you want to create an exception which will be utilized as a specialized case of any built-in exception, derive it from just this one. If you want to build your own hierarchy, and don't want it to be closely connected to Python's exception tree, derive it from any of the top exception classes, like Exception.

In the code sample below we've defined our own exception, named **MyZeroDivisionError**, derived from the built-in **ZeroDivisionError**. As you can see, we've decided not to add any new components to the class. In effect, an exception of this class can be – depending of the desired point of view – treated like a plain ZeroDivisionError, or considered separately. The **doTheDivision()** function raises either a **MyZeroDivisionError** or **ZeroDivisionError** exception, depending on the argument's value. The function is invoked four times in total, while the first two invocations are handled using only one except branch (the more general one) and the last two ones

with two different branches, able to distinguish the exceptions (don't forget: the order of the branches makes a fundamental difference!)

```
1  class MyZeroDivisionError(ZeroDivisionError):
2          pass
3
4  def doTheDivision(mine):
5          if mine:
6                  raise MyZeroDivisionError("worse news")
7          else:
8                  raise ZeroDivisionError("bad news")
9
10 for mode in [False, True]:
11         try:
12                 doTheDivision(mode)
13         except ZeroDivisionError:
14                 print('Division by zero')
15
16
17 for mode in [False, True]:
18         try:
19                 doTheDivision(mode)
20         except MyZeroDivisionError:
21                 print('My division by zero')
22         except ZeroDivisionError:
23                 print('Original division by zero')
```

Code Result

```
Division by zero
Division by zero
Original division by zero
My division by zero
```

4.11 Extending Exceptions Example

Here we've provided a good and yet simple example to help you understand extending exception from the scratch!

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self,message)
        self.pizza = pizza

class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self,pizza,message)
        self.cheese = cheese

def makePizza(pizza,cheese):
        if pizza not in ['margherita', 'capricciosa', 'calzone']:
                raise PizzaError(pizza,"no such pizza in menu")
        if cheese > 100:
                raise TooMuchCheeseError(pizza, cheese, "too much cheese")
        print("Pizza ready!")

for (pz,ch) in [('calzone',0),('margherita',110),('mafia',20)]:
        try:
                makePizza(pz,ch)
        except TooMuchCheeseError as tmce:
                print(tmce, ':', tmce.cheese)
        except PizzaError as pe:
                print(pe, ':', pe.pizza)
```

Code Result

```
Pizza ready!
too much cheese : 110
no such pizza in menu : mafia
```

5 Strings

5.1 Terminology

- Computers store characters as numbers. Every character used by a computer corresponds to a unique number, and vice versa.
- The word "internationalization" is commonly shortened to I18N → Why? Look carefully – there is an I at the front of the word, next there are 18 different letters, and an N at the end.
- A classic form of ASCII code uses eight bits for each sign. Eight bits mean 256 different characters. The first 128 are used for the standard Latin alphabet (both upper-case and lower-case characters).
- A **code point** is a number which makes a character. For example, 32 is a code point which makes a space in ASCII encoding. We can say that standard ASCII code consists of 128 code points.
- A **code page** is a standard for using the upper 128 code points to store specific national characters. This means that the one and same code point can make different characters when used in different code pages.
- In consequence, to determine the meaning of a specific code point, you have to know the target code page.
- **Unicode** assigns unique (unambiguous) characters (letters, hyphens, ideograms, etc.) to more than a million code points.
- The first 128 Unicode code points are identical to ASCII, and the first 256 Unicode code points are identical to the ISO/IEC 8859-1 code page (a code page designed for western European languages).
- **UCS-4 uses 32 bits** (four bytes) to store each character, and the code is just the Unicode code points' unique number. A file containing UCS-4 encoded text may start with a BOM (byte order mark), an unprintable combination of bits announcing the nature of the file's contents. Some utilities may require it. As you can see, UCS-4 is a rather wasteful standard – it increases a text's size by four times compared to standard ASCII.
- **UTF-8** The name is derived from Unicode Transformation Format.The concept is very smart. UTF-8 uses as many bits for each of the code

points as it really needs to represent them. For example all Latin characters (and all standard ASCII characters) occupy eight bits; and non-Latin characters occupy 16 bits and non-Latin characters occupy 16 bits;

- **Python3 is completely I18Ned** because it fully supports Unicode and UTF-8:
  - you can use Unicode/UTF-8 encoded characters to name variables and other entities;
  - you can use them during all input and output.
- Don't forget that a backslash (\) used as an escape character is **not included** in the string's total length.
- Python's strings are **immutable sequences**. so:
  - `del text[0]` does not work. The only thing you can do with `del` and a string is to remove the string as a whole.
  - Obviously you can't use `text.append('g')`, as the same for `insert()`
  - Don't think that a string's immutability limits your ability to operate with strings. The only consequence is that you have to remember about it, and implement your code in a slightly different way – look at the example:

```
1 text = "fgakjgkasdjgaga"
2 text = "A" + text
3 text = text + "Z"
4 print(text)
```

Code Result

```
AfgakjgkasdjgagaZ
```

5.2 Multiline string

The string has to start with three apostrophes, not one. The same tripled apostrophe is used to terminate it. The number of text lines put inside such a string is arbitrary. The multiline strings can be delimited by triple quotes, too. The code below return 15 as result, because `\n` will count in length in python

```
MultiLine = '''Line #1
Line #2'''

print('The string length is: ', end='')
print(len(MultiLine))
```

Code Result

```
The string length is: 15
```

5.3 comparing

- The final relation between strings is determined by comparing the first different character in both strings (keep ASCII/UNICODE code points in mind at all times)
- When you compare two strings of different lengths and **the shorter one is identical to the longer one's beginning**, the longer string is considered greater.
- Even if a string contains digits only, it's still not a number. It's interpreted as-is, like any other regular string, and its (potential) numerical aspect is not taken into consideration in any way.
- Comparing strings against numbers is generally a bad idea. The only comparisons you can perform with impunity are these symbolized by the `=` and `!` operators. The former always gives False, while the latter always produces True. Using any of the remaining comparison operators will raise a TypeError exception.

5.4 Operations

5.4.1 concatenated

The + operator used against two or more strings produces a new string containing all the characters from its arguments (note: the order matters – this overloaded +, in contrast to its numerical version, is not commutative)

5.4.2 Replicated

the * operator needs a string and a number as arguments; in this case, the order doesn't matter – you can put the number before the string, or vice versa, the result will be the same – a new string created by the nth replication of the argument's string.

5.4.3 ord()

If you want to know a specific character's ASCII/UNICODE code point value, you can use a function named ord() (as in ordinal).

5.4.4 chr()

If you know the code point (number) and want to get the corresponding character, you can use a function named chr().

5.4.5 slice

Moreover, everything you know about slices is still usable.

```
1 alpha = "abdefg"
2 print(alpha[1:3])
3 print(alpha[3:])
4 print(alpha[:3])
5 print(alpha[3:-2])
6 print(alpha[-3:4])
7 print(alpha[::2])
8 print(alpha[1::2])
```

Code Result

```
bd
efg
abd
e
e
adf
beg
```

5.4.6 min()

The function finds the minimum element of the sequence passed as an argument. There is one condition – the sequence (string, list, it doesn't matter) cannot be empty, or else you'll get a ValueError exception. According to ASCII table upper case alphabet less than normal case of an alphabet

```
t = 'The Knights Who Say Ni'
print('[' + min(t) + ']')
t = [ 0, 2, 3 ]
print(min(t))
```

Code Result

```
[K]
0
```

5.4.7 index()

The `index()` method (it's a method, not a function) searches the sequence from the beginning, in order to find the first element of the value specified in its argument. Note that the element searched for must occur in the sequence – its absence will cause a `ValueError` exception. exp: `print('sdfaksdgagalrkg'.index(k))`

5.4.8 list()

The list() function takes its argument (a string) and creates a new list containing all the string's characters, one per list element. Note that t's not

strictly a string function – list() is able to create a new list from many other entities (e.g., from tuples and dictionaries).

```
1 print(list('salam'))
```

Code Result

```
['s', 'a', 'l', 'a', 'm']
```

5.4.9 count()

The count() method counts all occurrences of the element inside the sequence. The absence of such elements doesn't cause any problems.

5.4.10 sorted() vs sort()

- **sorted()** The function takes one argument (a list) and returns a new list, filled with the sorted argument's elements.
- **sort()** affects the list itself – no new list is created. Ordering is performed in situ by the method named sort() so it's not useful for strings

5.5 String methods

Note: methods don't have to be invoked from within variables only. They can be invoked directly from within string literals. We're going to use that convention regularly

5.5.1 capitalize()

- if the first character inside the string is a letter (note: the first character is an element with an index equal to 0, not just the first visible character), it will be converted to upper-case;
- all remaining letters from the string will be converted to lower-case.
- the original string (from which the method is invoked) is not changed in any way
- the modified (capitalized in this case) string is returned as a result – if you don't use it in any way (assign it to a variable, or pass it to a function/method) it will disappear without a trace.

```
1 print('Alpha'.capitalize())
2 print('ALPHA'.capitalize())
3 print(' Alpha'.capitalize())
4 print('123'.capitalize())
5 print("".capitalize())
```

Code Result

```
Alpha
Alpha
 alpha
123
```

5.5.2 center()

- The one-parameter variant of the center() method makes a copy of the original string, trying to center it inside a field of a specified width. The centering is actually done by adding some spaces before and after the string. Don't expect this method to demonstrate any sophisticated

skills. It's rather simple. The example uses brackets to clearly show you where the centered string actually begins and terminates

- The two-parameter variant of center() makes use of the character from the second argument, instead of a space

```
print('['+'alpha'.center(1)+']')
print('['+'alpha'.center(30)+']')
print('['+'alpha'.center(10)+']')
print('['+'alpha'.center(10, '*')+']')
```

Code Result

```
[alpha]
[            alpha             ]
[  alpha   ]
[**alpha***]
```

5.5.3 endswith()

The endswith() method checks if the given string ends with the specified argument and returns True or False, depending of the check result.

```
t = 'zeta'
print(t.endswith('a'))
print(t.endswith('A'))
print(t.endswith('et'))
```

Code Result

```
True
False
False
```

5.5.4 startswitch()

The startswith() method is a mirror reflection of endswith() – it checks if a given string starts with the specified substring.

5.5.5 find()

- The find() method is similar to index(), which you already know (it looks for a substring), but: it's safer – it doesn't generate an error for an argument containing a non-existent substring (it returns -1 then) **it works with strings only** – don't try to apply it to any other sequence.
- If you want to perform the find, not from the string's beginning, but from any position, you can use a two-parameter variant of the find() method → The second argument specifies the index at which the search will be started (it doesn't have to fit inside the string).
- There is also a three-parameter mutation of the find() method – the third argument points to the first index which won't be taken into consideration during the search (it's actually the upper limit of the search) → The second argument specifies the index at which the search will be started (it doesn't have to fit inside the string)

don't use find() if you only want to check if a single character occurs within a string – the in operator will be significantly faster.

```
# if it finds the string, it will return the first index where the string appears

print("--------------------")
t = 'theta'
print(t.find('eta'))
print(t.find('ta'))
print(t.find('the'))
print(t.find('tlhe'))

print("--------------------")

print('kappa'.find('a',2))
print('kappa'.find('a', 2, 4))
print('kappa'.find('a',8))

print("--------------------")

txt = ''' A variation of the ordinary lorem ipsum text has been used in
typesetting since the 1960s or earlier, when it was popularized by
```

```
advertisements for Letraset transfer sheets. It was introduced to the
Information Age in the mid-1980s by Aldus Corporation, which employed
it in graphics and word-processing templates for its desktop
publishing program PageMaker.'''

fnd = txt.find('the')
while fnd != -1:
    print(fnd, end="' ")
    fnd = txt.find('the', fnd+1)
```

Code Result

```
--------------------
2
3
0
-1
--------------------
4
-1
-1
--------------------
16' 81' 196' 219'
```

5.5.6 rfind()

The one-, two-, and three-parameter methods named rfind() do nearly the same things as their counterparts (the ones devoid of the r prefix), but start their searches from the end of the string, not the beginning (hence the prefix r, for right).

```
t = 'tau tau tau'
tIndex = dict()

for i in range(len(t)):
    tIndex[i] = t[i]


for key in tIndex:
    print(key, ":", tIndex[key])
```

```
10
11
12 print("--------------------")
13 print(t.rfind('ta'))
14 print(t.rfind('ta',9))
15 print(t.rfind('ta',3,9))
```

Code Result

```
0 : t
1 : a
2 : u
3 :
4 : t
5 : a
6 : u
7 :
8 : t
9 : a
10 : u
--------------------
8
-1
4
```

5.5.7 isalnum()

The parameterless method named isalnum() checks if the string contains only digits or alphabetical characters (letters), and returns True/False according to the result →

Note: any string element that is not a digit or a letter causes the method to return False. An empty string does, too.

```
1 print('is all'.isalnum())
2 print('10E4'.isalnum())
3 print(''.isalnum())
```

Code Result

```
False
True
```

```
False
```

5.5.8 isalpha()

The isalpha() method is more specialized – it's interested in letters only

5.5.9 isdigit()

In turn, the isdigit() method looks at digits only – anything else produces False as the result.

5.5.10 islower()

The islower() method is a fussy variant of isalpha() – it accepts lower-case letters only.

5.5.11 isupper()

The isupper() is the upper-case version of islower() – it concentrates on upper-case letters only.

5.5.12 isspace()

The isspace() method identifies whitespaces only – it disregards any other character (the result is False then). Note that `\n` is a whitespace

5.5.13 join()

The join() method is rather complicated, so let us guide you step by step thorough it:

- as its name suggests, the method performs a join – it expects one argument as a list; it must be assured that all the list's elements are strings – the method will raise a TypeError exception otherwise;
- all the list's elements will be joined into one string but ...

- the string from which the method has been invoked is used as a separator, put among the strings; it can be empty or whitespace also or arbitrary long
- the newly created string is returned as a result.

```
print(','.join(['omicron','pi','rho']))
print(' '.join(['omicron','pi','rho']))
```

Code Result

```
omicron,pi,rho
omicron pi rho
```

5.5.14 lower()

The lower() method makes a copy of a source string, replaces all upper-case letters with their lower-case counterparts, and returns the string as the result. Again, the source string remains untouched.

```
print('LghWg60'.lower())
```

Code Result

```
lghwg60
```

5.5.15 upper()

The upper() method makes a copy of the source string, replaces all lower-case letters with their upper-case counterparts, and returns the string as the result.

```
print('SiGmA=60'.upper())
```

Code Result

```
SIGMA=60
```

5.5.16 swapcase()

The swapcase() method makes a new string by swapping the case of all letters within the source string: lower-case characters become upper-case, and vice

versa.

```
print('One thing I know, that I know nothing'.swapcase())
```

Code Result

```
oNE THING i KNOW, THAT i KNOW NOTHING
```

5.5.17 title()

The title() method performs a somewhat similar function – it changes every word's first letter to upper-case, turning all other ones to lower-case.

```
print('One thing I know, that I know nothing'.title())
```

Code Result

```
One Thing I Know, That I Know Nothing
```

5.5.18 lstrip() MORE_STUDY

- The parameterless lstrip() method returns a newly created string formed from the original one by removing all leading whitespaces.
- The one-parameterlstrip() method does the same as its parameterless version, but removes all characters enlisted in its argument (a string), not just whitespaces

```
print('['+' tau '.lstrip()+']')
print('www.cisco.com'.lstrip('w.'))
```

Code Result

```
[tau ]
cisco.com
```

5.5.19 rstrip() MORE_STUDY

Two variants of the rstrip() method do nearly the same as lstrips, but affect the opposite side of the string.

```
print('['+' upsilon '.rstrip()+']')
print('www.cisco.com'.rstrip('.com'))
print('www.cisco.com'.rstrip('com'))
print('www.cisco.com'.rstrip('m'))
```

Code Result

```
[ upsilon]
www.cis
www.cisco.
www.cisco.co
```

5.5.20 strip()

The strip() method combines the effects caused by rstrip() and lstrip() – it makes a new string lacking all the leading and trailing whitespaces.

```
print('['+'  aleph  '.strip()+']')
```

Code Result

```
[aleph]
```

5.5.21 replace()

- The two-parameter method replace() returns a copy of the original string in which all occurrences of the first argument have been replaced by the second argument
- The three-parameter replace() variant uses the third argument (a number) to limit the number of replacements

```
print('this is it, these is it'.replace('is','are',2))
```

Code Result

```
thare are it, these is it
```

5.5.22 split()

The split() method does what it says – it splits the string and builds a list of all detected substrings. The method assumes that the substrings are delimited by whitespaces – the spaces don't take part in the operation, and aren't copied into the resulting list. If the string is empty, the resulting list is empty too.

```
1 print('phi   chi\npsi'.split())
```

Code Result

```
['phi', 'chi', 'psi']
```

6 Object Oriented Programming

6.1 Terminology

- The object approach suggests a completely different way of thinking. The data and the code are enclosed together in the same world, divided into classes.
- Every class is like a recipe which can be used when you want to create a useful object (this is where the name of the approach comes from). You may produce as many objects as you need to solve your problem.
- Every object has a set of traits (they are called properties or attributes – we'll use both words synonymously) and is able to perform a set of activities (which are called methods).
- The recipes may be modified if they are inadequate for specific purposes and, in effect, new classes may be created. These new classes inherit properties and methods from the originals, and usually add some new ones, creating new, more specific tools.
- The objects interact with each other, exchanging data or activating their methods. A properly constructed class (and thus, its objects) are able to protect the sensible data and hide it from unauthorized modifications. There is no clear border between data and code: they live as one in objects.
- A **class** (among other definitions) is a set of objects.
- An **object** is a being belonging to a class. An object is an incarnation of the requirements, traits, and qualities assigned to a specific class.
- **Classes form a hierarchy**. This may mean that an object belonging to a specific class belongs to all the superclasses at the same time. It may also mean that any object belonging to a superclass may not belong to any of its subclasses.
- Each subclass is more specialized (or more specific) than its superclass. Conversely, each superclass is more general (more abstract) than any of its subclasses.

6.2 Inheritance

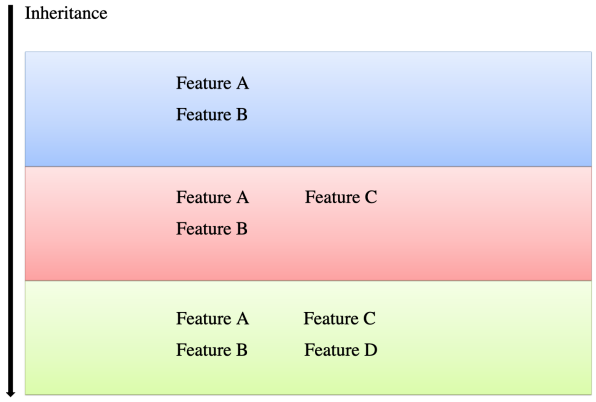

Figure 5: OOP inheritance

- Let's define one of the fundamental concepts of object programming, named "inheritance". Any object bound to a specific level of a class hierarchy inherits all the traits (as well as the requirements and qualities) defined inside any of the superclasses. The object's home class may define new traits (as well as requirements and qualities) which will be inherited by any of its superclasses.
- Inheritance is a common practice (in object programming) of passing attributes and methods from the superclass (defined and existing) to a newly created class, called the subclass.
- The most important factor of the process is the relation between the superclass and all of its subclasses (note: if B is a subclass of A and C is a subclass of B, this also means than C is a subclass of A, as the relationship is fully transitive).

6.3 Objects Roles

1. an object has a **name** that uniquely identifies it within its home namespace (although there may be some anonymous objects, too)
2. an object has a set of individual **properties** which make it original, unique or outstanding (although it's possible that some objects may have no properties at all)

3. an object has a set of abilities to perform specific **activities**, able to change the object itself, or some of the other objects.

6.4 Classes

- The class you define has nothing to do with the object: the existence of a class does not mean that any of the compatible objects will automatically be created. The class itself isn't able to create an object – you have to create it yourself, and Python allows you to do this.
- The definition begins with the keyword class. The keyword is followed by an identifier which will name the class (note: don't confuse it with the object's name – these are two different things). Next, you add a colon, as classes, like functions, form their own nested block. The content inside the block define all the class's properties and activities.
- The `pass` keyword fills the class with nothing. It doesn't contain any methods or properties.

```
1 class OurClass:
2     pass
```

6.5 Object Instantiation

- The newly defined class becomes a tool that is able to create new objects.
- The tool has to be used explicitly, on demand.
- Imagine that you want to create one (exactly one) object of the `OurClass` class: To do this, you need to assign a variable to store the newly created object of that class, and create an object at the same time. Note that the class name tries to pretend that it's a function
- The act of creating an object of the selected class is also called an **instantiation** (as the object becomes an instance of the class).

```
1 ourObject = ourClass()
```

6.5.1 Constructor

A constructor is a special kind of method that Python calls when it instantiates an object using the definitions found in your class. Python relies on the constructor to perform tasks such as initializing (assigning values to) any instance variables that the object will need when it starts. Constructors can also verify that there are enough resources for the object and perform any other start-up task you can think of.

- The name of a constructor is always the same, `__init__()`. The constructor can accept arguments when necessary to create the object. When you create a class without a constructor, Python automatically creates a default constructor for you that doesn't do anything. Every class must have a constructor, even if it simply relies on the default constructor.
- It has to have at least **one parameter** (we'll discuss this later); the parameter is used to represent the newly created object – you can use the parameter to manipulate the object, and to enrich it with the needed properties;
- the obligatory parameter is usually named `self` – it's only a convention, but you should follow it – it simplifies the process of reading and understanding your code.
- we've used the dotted notation, just like when invoking methods; this is the general convention for accessing an object's properties – you need to name the object, put a dot after it, and specify the desired property's name; don't use parentheses! You don't want to invoke a method – you want to access a property;
- if you set a property's value for the very first time (like in the constructor), you are creating it; from that moment on, the object has got the property and is ready to use its value;

```
1 class MyClass:
2     def __init__(self, Name="there"):
3         self.Greeting = Name + "!"
4 
5     def SayHello(self):
6             print("Hello {0}".format(self.Greeting))
7 
```

```
8
9  myInstance = MyClass()
10 myInstance.SayHello()
11
12 shaeInstance = MyClass("SHAE")
13 shaeInstance.SayHello()
```

Code Result

```
Hello there!
Hello SHAE!
```

6.5.2 Encapsulation

When any class component has a name **starting with two underscores**, it becomes **private** – this means that it can be accessed only from within the class. You cannot see it from the outside world. This is how Python implements the encapsulation concept.

6.5.3 Methods

- All methods have to have a `self` parameter as their first parameter(The self name is only a convention but it's existence is necessary). It plays the same role as the first constructor parameter.
- It allows the method to access entities (properties and activities/methods) carried out by the actual object. **You cannot omit it**. Every time Python invokes a method, **it implicitly sends the current object as the first argument.**
- If you want to make a method **private** you have to name it **starting with two underscores** otherwise the method will be **public**.
- There is one more requirement – the name must have no more than one trailing underscore. As no trailing underscores at all fully meets the requirement, you can assume that the name is acceptable.

6.6 Stack

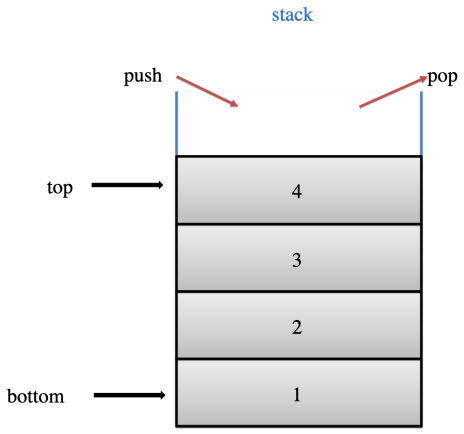

Figure 6: Stack

- A stack is a structure developed to store data in a very specific way. Imagine a stack of coins. You aren't able to put a coin anywhere else but on the top of the stack. Similarly, you can't get a coin off the stack from any place other than the top of the stack. If you want to get the coin that lies on the bottom, you have to remove all the coins from the higher levels.
- The alternative name for a stack (but only in IT terminology) is **LIFO**. It's an abbreviation for a very clear description of the stack's behavior: "Last In – First Out". The coin that came last onto the stack will leave first.
- A stack is an object with two elementary operations, conventionally named `push` (when a new element is put on the top) and `pop` (when an existing element is taken away from the top).
- Stacks are used very often in many classical algorithms, and it's hard to imagine the implementation of many widely used tools without the use of stacks.

6.6.1 Procedural Stack

6.6.1.1 push - pop

Here we will define a stack and then `push` and `pop` functions. Note that `puah` function doesn't return anything and `pop` function doesn't check if there is any element in the stack.

```
stack = []

def push(val):
    stack.append(val)

def pop():
    val = stack[-1]
    del stack[-1]
    return val

push(2)
push(2)
push(1)
print(pop())
print(pop())
print(pop())
```

6.6.2 Objective Stack

The objective approach delivers very essential pros to stacks like:

- the ability to **hide** (protect) selected values against unauthorized access is called **encapsulation**; the encapsulated values can be neither accessed nor modified if you want to use them exclusively;
- when you have a class implementing all the needed stack behaviors, you can **produce as many stacks as you want**; you needn't copy or replicate any part of the code;
- the ability to enrich the stack with new functions comes from **inheritance**; you can create a new class (a subclass) which inherits all the existing traits from the superclass, and adds some new ones.

6.6.2.1 push - pop

Having such a class opens up some new possibilities. For example, you can now have more than one stack behaving in the same

way. Each stack will have its own copy of private data, but will utilize the same set of methods.

```
# This is the class itself
class Stack:
        def __init__(self):
                self.__stk = []

        def push(self, val):
                self.__stk.append(val)

        def pop(self):
                val = self.__stk[-1]
                del self.__stk[-1]
                return val
```

```
# This is noweb syntax to impart class on org-mode
<<objectiveStack>>

stack = Stack()
stack.push(3)
stack.push(2)
stack.push(1)
print(stack.pop())
print(stack.pop())
print(stack.pop())
print("------------------")
stack1 = Stack()
stack2 = Stack()
stack1.push(3)
stack2.push(stack1.pop())
print(stack2.pop())
```

Code Result

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

6.7 Subclasses

- We don't want to modify the previously defined stack. It's already good enough in its applications, and we don't want it changed in any way. **We want a new stack with new capabilities.** In other words, we want to construct a subclass of the already existing Stack class.
- The first step is easy: just define a new subclass pointing to the class which will be used as the superclass.

```
class addingStack(Stack)
```

- The class doesn't define any new component yet, but that doesn't mean that it's empty. **It gets all the components defined by its superclass**
- Contrary to many other language, **Python forces you to explicitly invoke a superclass' constructor**. Omitting this point will have harmful effects.

```
1 class addingStack(Stack):
2     def __init__(self):
3         Stack.__init__(self)
4         self.__sum = 0
```

- Note the syntax above:
  - you specify the **superclass's name** (this is the class whose constructor you want to run)
  - you put a dot after it;
  - you specify the name of the constructor;
  - you have to point to the object (the class's instance) which has to be initialized by the constructor – this is why you have to specify the argument and use the self variable here; note: invoking any method (including constructors) from outside the class never requires you to put the self argument at the argument's list – **invoking a method from within the class demands explicit usage of the `self` argument,** and it has to be put first on the list.

6.7.1 Changing functionality of methods

```
def push(self, val):
    self.__sum += val
    Stack.push(self, val)
```

Is it really adding? We have these methods in the superclass already. Can we do something like that? Yes, we can. It means that we're going to **change the functionality of the methods, not their names.** We can say more precisely that the interface (the way in which the objects are handled) of the class remains the same when changing the implementation at the same time.

Note: the second activity has already been implemented inside the superclass – so we can use that. Furthermore, we have to use it, as there's no other way to access the `__stk` variable.

Note the way we've invoked the previous implementation of the push method (the one available in the superclass):

- we have to specify the superclass's name; this is necessary in order to clearly indicate the class containing the method, to avoid confusing it with any other function of the same name;
- we have to specify the target object and to pass it as the first argument (it's not implicitly added to the invocation in this context)

We say that the `push` method has been overridden – the same name as in the superclass now represents a different functionality.

```
# This is noweb syntax to impart class on org-mode
<<objectiveStack>>

class AddingStack(Stack):
        def __init__(self):
                Stack.__init__(self)
                self.__sum = 0

        def getSum(self):
                return self.__sum

        def push(self, val):
                self.__sum += val
```

```
14                  Stack.push(self,val)
15
16          def pop(self):
17                  val = Stack.pop(self)
18                  self.__sum -= val
19                  return val
20
21  stack = AddingStack()
22  for i in range(5):
23          stack.push(i)
24  print(stack.getSum())
25  print("------------------")
26  for i in range(5):
27          print(stack.pop())
```

Code Result

```
10
------------------
4
3
2
1
0
```

6.8 Instance variable

Object-oriented programming allows for variables to be used at the **class level** or the **instance level**. Variables are essentially symbols that stand in for a value you're using in a program. At the class level, variables are referred to as class variables, whereas variables at the instance level are called instance variables.

When we expect variables are going to be consistent across instances, or when we would like to initialize a variable, we can define that variable at the class level. When we anticipate the variables will change significantly across instances, we can define them at the instance level.

Python objects, when created, are gifted with a small set of predefined properties and methods. Each object has got them, whether you want them or not. One of them is a variable named `__dict__` (it's a dictionary). The

variable contains the names and values of all the properties (variables) the object is currently carrying. Let's make use of it to safely present an object's contents.

```
class Class:
        def __init__(self, val=1):
                self.First = val

        def setSecond(self, val=2):
                self.Second = val


object1 = Class()
object2 = Class(2)
object2.setSecond(3)
object3 = Class(4)
object3.Third = 5


print(object1.__dict__)
print(object2.__dict__)
print(object3.__dict__)
```

Code Result

```
{'First': 1}
{'First': 2, 'Second': 3}
{'First': 4, 'Third': 5}
```

- the class named Class has a constructor, which unconditionally creates an instance variable named First, and sets it with the value passed through the first argument (from the class user's perspective) or the second argument (from the constructor's perspective); note the default value of the parameter – any trick you can do with a regular function parameter can be applied to methods, too;
- the class also has a method which creates another instance variable, named Second;
- we've created three objects of the class Class, but all these instances differ:
  - object1 only has the property named First;

- object2 has two properties: First and Second;
- object3 has been enriched with a property named Third just on the fly, outside the class's code – this is possible and fully permissible.

Important: modifying an instance variable of any object has **no impact** to all the remaining objects. Instance variables are perfectly isolated from each other.

6.8.1 mangling

In Python, mangling is used for "private" class members which are designated as such by giving them a name with two leading underscores and no more than one trailing underscore.

```
1  class Class:
2          def __init__(self, val=1):
3                  self.__First = val
4
5          def setSecond(self, val=2):
6                  self.__Second = val
7
8
9  object1 = Class()
10 object2 = Class(2)
11 object2.setSecond(3)
12 object3 = Class(4)
13 object3.__Third = 5
14
15
16 print(object1.__dict__)
17 print(object2.__dict__)
18 print(object3.__dict__)
19 print(object1._Class__First)
```

Code Result

```
{'_Class__First': 1}
{'_Class__First': 2, '_Class__Second': 3}
{'_Class__First': 4, '__Third': 5}
1
```

It's nearly the same as the previous one. The only difference is in the property names. We've added two underscores in front of them. As you know, such an addition makes the instance variable private – it becomes inaccessible from the outer world.

When Python sees that you want to add an instance variable to an object and you're going to do it inside any of the object's methods, it **mangles** the operation in the following way:

- it puts a class name before your name;
- it puts an additional underscore at the beginning.

This is why the `__First` becomes `_Class__First`. The name is now fully accessible from outside the class. You can run a code like this:

- `print(object1._Class__First)`

6.9 Class variable

A class variable is a property which exists in just one copy and is stored outside any object. **initializing the variable inside the class but outside any of its methods makes the variable a class variable;**

```
class Class:

        Counter = 0

        def __init__(self, val=1):
                self.__First = val
                Class.Counter += 1


object1 = Class()
print(object1.__dict__, object1.Counter)
print("--------------------")

object2 = Class(2)
object3 = Class(4)
print(object1.__dict__, object1.Counter)
print(object2.__dict__, object2.Counter)
print(object3.__dict__, object3.Counter)
```

Code Result

```
{'_Class__First': 1} 1
--------------------
{'_Class__First': 1} 3
{'_Class__First': 2} 3
{'_Class__First': 4} 3
```

Notes:

- class variables aren't shown in an object's `__dict__` (this is natural as class variables aren't parts of an object) but you can always try to look into the variable of the same name, but at the class level – we'll show you this very soon;
- a class variable always presents the same value in all class instances (objects)

Mangling a class variable's name has the same effects as those you're already familiar with.

```
class Class:
        __Counter = 0

        def __init__(self, val=1):
                self.__First = val
                Class.__Counter += 1


object1 = Class()
object2 = Class(2)
object3 = Class(4)

print(object1.__dict__, object1._Class__Counter)
print(object2.__dict__, object2._Class__Counter)
print(object3.__dict__, object3._Class__Counter)
```

Code Result

```
{'_Class__First': 1} 3
{'_Class__First': 2} 3
{'_Class__First': 4} 3
```

6.10 hasattr

Python provides a function which is able to safely check if any object/class contains a specified property. The function is named hasattr, and expects two arguments to be passed to it:

- the **class or the object** being checked;
- the **name of the property** whose existence has to be reported (*note: it has to be a string containing the attribute name, not the name alone*)
- Don't forget that the hasattr() function can operate on classes, too. You can use it to find out if a class variable is available

```
1  class Class:
2          Attr = 1
3
4          def __init__(self, val):
5                  if val % 2 != 0:
6                          self.a = 1
7                  else:
8                          self.b = 2
9
10
11 object = Class(1)
12 if hasattr(object, 'a'):
13         print(object.a)
14 else:
15         print(object.b)
16
17 print("--------------------")
18
19 print(hasattr(Class, 'Attr'))
20 print(hasattr(Class, 'pop'))
```

Code Result

```
1
--------------------
True
False
```

Another useful example:

```
class Class:
        a = 1

        def __init__(self):
                self.b = 2


object = Class()
print(hasattr(object,'b'))
print(hasattr(object,'a'))
print(hasattr(Class,'b'))
print(hasattr(Class,'a'))
```

Code Result

```
True
True
False
True
```

6.11 Classes methods

A method is a function embedded inside a class. There is one fundamental requirement – a method is obliged to have at least one parameter (there are no such thing as parameterless method – a method may be invoked without an argument, but not declared without parameters). The first (or only) parameter is usually named `self`. The name `self` suggests the parameter's purpose – it identifies the object for which the method is invoked.

- The `self` parameter is used to obtain access to the object's instance and class variables.

```
class Class:
    Variable = 3

    def method(self):
        print(self.Variable, self.var)


obj = Class()
obj.var = 4
obj.method()
```

Code Result

```
3 4
```

- The `self` parameter is also used to invoke other object/class methods from inside the class.

```
class Class:
    def other(self):
        print("Other")

    def method(self):
        print("Method")
        self.other()


obj = Class()
```

```
11 obj.method()
```

Code Result

```
Method
Other
```

- If you name a method `__init__`, it won't be a regular method – **it will be a constructor.** If a class has a constructor, *it is invoked automatically and implicitly when the object of the class is instantiated.*

  Note that the constructor:

  - cannot return a value,as it is designed to return a newly created object and nothing else;
  - cannot be invoked directly either from the object or from inside the class (you can invoke a constructor from any of the object's superclasses, but we'll discuss this issue later).

```
1 class Class:
2
3     def __init__(self, val):
4         self.var = val
5
6
7 obj = Class(5)
8 print(obj.var)
```

Code Result

```
5
```

- Everything we've said about property name mangling applies to method names, too – a method whose name starts with `__` is (partially) hidden.

```
1 class Class:
2
```

```
    def visible(self):
        print("Visible")

    def __hidden(self):
        print("Hidden")


obj = Class()
obj.visible()

try:
    obj.__hidden()
except:
    print("Failed")

obj._Class__hidden()
```

Code Result

```
Visible
Failed
Hidden
```

6.12 Introspection/Reflection

- **Introspection**, which is the ability of a program to examine the type or properties of an object at runtime;
- **Reflection**, which goes a step further, and is the ability of a program to manipulate the values, properties and/or functions of an object at runtime.

In other words, you don't have to know a complete class/object definition to manipulate the object, as the object and/or its class contain the metadata allowing you to recognize its features during program execution.

6.12.1 `__name__`

The property contains the name of the class. It's a string. the `__name__` attribute is absent from the object – **it exists only inside classes.**

If you want to find the class of a particular object, you can use a function named `type()`, which is able (among other things) to find a class which has been used to instantiate any object.

```
class Classy:
        pass


print(Classy.__name__)
obj = Classy()
print(type(obj).__name__)
```

Code Result

```
Classy
Classy
```

6.12.2 `__module__`

`__module__` is a string, too – it stores the name of the module which contains the definition of the class.

```
class Classy:
    pass


print(Classy.__module__)
obj = Classy()
print(obj.__module__)
```

Code Result

```
__main__
__main__
```

As you know, any module named `__main__` is actually not a module, but the file currently being run.

6.12.3 __base__

__bases__ is a tuple. The tuple contains classes (not class names) which are direct superclasses for the class. The order is the same as that used inside the class definition.

```
class Super1:
        pass


class Super2:
        pass


class Sub(Super1, Super2):
        pass


def printbases(cls):
        print('( ', end='')
        for x in cls.__bases__:
                print(x.__name__, end=' ')
        print(')')


printbases(Super1)
printbases(Super2)
printbases(Sub)
```

Code Result

```
( object )
( object )
( Super1 Super2 )
```

6.13 Internal object identifier

if you run the code

```
print(objectName)
```

then you will see something like

`<__main__.Star object at 0x7f1074cc7c50>`

in which the `0x7f1074cc7c50` is the **internal object identifier**

6.14 `__str__`

When Python needs any class/object to be presented as a string (putting an object as an argument in the `print()` function invocation fits this condition) it tries to invoke a method named `__str__()` from the object and to use the string it returns. The default `__str__()` method returns something like `<__main__.Star object at 0x7f1074cc7c50>` – ugly and not very informative. You can change it just by defining your own method of the name.

```
1  class Star:
2          def __init__(self,  name,  galaxy):
3                  self.name = name
4                  self.galaxy = galaxy
5
6          def __str__(self):
7                  return self.name + ' in ' + self.galaxy
8
9
10 sun = Star("Sun", "Milky Way")
11 print(sun)
```

Code Result

```
Sun in Milky Way
```

6.15 issubclass

Python offers a function which is able to identify a relationship between two classes, and although its diagnosis isn't complex, it can check if a particular class is a subclass of any other class.

```
issubclass(class1, class2)
```

The function returns True if class1 is a subclass of class2, and False otherwise.

There is one important observation to make: each class is considered to be a subclass of itself.

```
1  class Vehicle:
2          pass
3
4
5  class LandVehicle(Vehicle):
6          pass
7
8
9  class TrackedVehicle(LandVehicle):
10         pass
11
12
13 for cl1 in [Vehicle, LandVehicle, TrackedVehicle]:
14         for cl2 in [Vehicle, LandVehicle, TrackedVehicle]:
15                 print(issubclass(cl1,cl2),end='\t')
16         print()
```

Code Result:

↓ is a subclass of →	Vehicle	LandVehicle	TrackedVehicle
Vehicle	**True**	**False**	**False**
LandVehicle	**True**	**True**	**False**
TrackedVehicle	**True**	**True**	**True**

Figure 7: Result table

6.16 instance vs variables

Variables don't store the objects themselves, but only the handles pointing to the internal Python memory. Assigning a value of an object variable to another variable doesn't copy the object, but only its handle. This is why an operator like `is` may be very useful in particular circumstances. he operator checks whether two variables (object1 and object2 here) refer to the same object.

```
object1 is object2
```

Sample code:

```
class ThisIsClass:
        def __init__(self, val):
                self.val = val


ob1 = ThisIsClass(0)
ob2 = ThisIsClass(2)
ob3 = ob1
ob3.val += 1
print(ob1 is ob2)
print(ob2 is ob3)
print(ob3 is ob1)
print(ob1.val, ob2.val, ob3.val)

str1 = "Mary had a little "
str2 = "Mary had a little lamb"
str1 += "lamb"
print(str1 == str2, str1 is str2)
```

Code Result

```
False
False
True
1 2 1
True False
```

- **Code Description**: There is a very simple class equipped with a simple constructor, creating just one property. The class is used to instantiate two objects. The former is then assigned to another variable, and its val property is incremented by one. Afterward, the is operator is applied three times to check all possible pairs of objects, and all val property values are also printed. The last part of the code carries out another experiment. After three assignments, both strings contain the same texts, but these texts are stored in different objects.

6.17 Override

See the code below:

```
1  class Level0:
2          Var = 100
3
4          def fun(self):
5                  return 101
6
7  class Level1(Level0):
8          Var = 200
9
10         def fun(self):
11                 return 201
12
13 class Level2(Level1):
14         pass
15
16
17 object = Level2()
18 print(object.Var, object.fun())
```

Code Result

```
200 201
```

Code Description: Both, Level0 and Level1 classes define a method named **fun()** and a property named **Var**. As you can see, the Var class variable and **fun()** method from Level1 class **override the entities of the same names** derived from the Level0 class.

6.18 How python treat classes

- When you try to access any object's entity, Python will try (in this order):
  1. to find it inside the object itself;
  2. to find it in all classes involved in the object's inheritance line from bottom to top;
  3. if both of the above fail, an exception (**AttributeError**) is raised.
- If a subClass has more than one superClass, python treats it in this order:
  1. inside the object itself;
  2. in its superclasses, from bottom to top;
  3. if there is more than one class on a particular inheritance path, Python scans them from left to right.

```
1  class Level0:
2          Var = 100
3
4          def fun(self):
5                  return 101
6
7
8  class Level1(Level0):
9          Var = 200
10
11         def fun(self):
12                 return 201
13
14
15 class Level2(Level1):
16         pass
17
18
19 object = Level2()
20 print(object.Var, object.fun())
```

Code Result

```
200 201
```

6.19 polymorphism

the situation in which the subclass is able to modify its superclass behavior (just like in the example below) is called polymorphism. The word comes from Greek (polys: "many, much" and morphē, "form, shape"), which means that **one and the same class can take various forms** depending on the redefinitions done by any of its subclasses. The method, redefined in any of the superclasses, thus changing the behavior of the superclass, is called **virtual**. In other words, **no class is given once and for all**. Each class's behavior may be modified at any time by any of its subclasses.

Example: There are two classes, named **One** and **Two**, while **Two** is derived from **One**. Nothing special. However, one thing looks remarkable – the **doit()** method. It's defined twice: originally inside **One** and subsequently inside **Two**. The essence of the example lies in the fact that it is invoked just once – inside **One**. **The second invocation will launch `doit()` in the form existing inside the `Two`.**

```
1  class One:
2          def doit(self):
3                  print("doit from One")
4          def doanything(self):
5                  self.doit()
6
7
8  class Two(One):
9          def doit(self):
10                 print("doit from Two")
11
12
13 one = One()
14 two = Two()
15
16 one.doanything()
17 two.doanything()
```

Code Result

```
doit from One
doit from Two
```

6.20 Composition

Composition is the process of **composing an object using other different objects**. The objects used in the composition deliver a set of desired traits (properties and/or methods) so we can say that they act like blocks used to build a more complicated structure. It can be said that:

- **Inheritance** extends a class's capabilities by adding new components and modifying existing ones; in other words, the complete recipe is contained inside the class itself and all its ancestors; the object takes all the class's belongings and makes use of them;
- **Composition** projects a class as a container able to store and use other objects (derived from other classes) where each of the objects implements a **part of a desired class's behavior.**

```
1  import time
2
3
4  class Tracks:
5          def changedirection(self, left, on):
6                  print("tracks: ", left, on)
7
8
9  class Wheels:
10         def changedirection(self, left, on):
11                 print("wheels: ", left, on)
12
13
14 class Vehicle:
15         def __init__(self, controller):
16                 self.controller = controller
17
18         def turn(self, left):
19                 self.controller.changedirection(left, True)
```

```
                time.sleep(0.25)
                self.controller.changedirection(left, False)


wheeled = Vehicle(Wheels())
tracked = Vehicle(Tracks())

wheeled.turn(True)
tracked.turn(False)
```

Code Result

```
wheels:  True True
wheels:  True False
tracks:  False True
tracks:  False False
```

7 Working with files

7.1 Addressing

Python is smart enough to be able to convert slashes into backslashes each time it discovers that it's required by the OS. This means that any the following assignments will work with Windows, too:

```
name = "/dir/file"

name = "c:/dir/file"
```

7.2 Open Modes

To connect (bind) the stream with the file, it's necessary to perform an explicit operation. The operation of connecting the stream with a file is called opening the file, while disconnecting this link is named closing the file. Hence, the conclusion is that the very first operation performed on the stream is always open and the last one is close. The program, in effect, is free to manipulate the stream between these two events and to handle the associated file. The opening of the stream is not only associated with the file, but should also declare the manner in which the stream will be processed. This declaration is called an **open mode.** There are three basic modes used to open the stream:

- **read mode:** a stream opened in this mode allows read operations only; trying to write to the stream will cause an exception (the exception is named `UnsupportedOperation`, which inherits `OSError` and `ValueError`, and comes from the `io` module); the file associated with the stream must exist and has to be readable, otherwise open() function raises exception.
- **write mode:** a stream opened in this mode allows write operations only; attempting to read the stream will cause the exception mentioned above; the file associated with the stream doesn't need to exist; if it doesn't exist it will be created; if it exists it will truncated to the length of zero (erased); if the creation isn't possible (e.g. due to system permissions) the open() function raises an exception
- **update mode**: a stream opened in this mode allows both writes and reads. the file associated with the stream doesn't need to exist; if it

doesn't exist it will be created; if it exists the virtual recording head will be set at the end of the file (the previous content of the file remains untouched)

Notes:

- If you open the stream with `r+`:
  - the stream will be opened in "read and update" mode;
  - The file associated with the stream must exist and has to be writeable, otherwise the open() function raises an exception
  - both read and write operations are allowed for the stream
- If you open the stream with `w+`:
  - the stream will be opened in "write and update" mode;
  - the file associated with the stream doesn't need to exist; if it doesn't exist it will be created; the previous content of the file remains untouched
  - both read and write operations are allowed for the stream

7.3 I/O

When you read something from a stream, a virtual head moves over the stream according to the number of bytes transferred from the stream. When you write something to the stream, the same head moves along the stream recording the data from the memory. Whenever we talk about reading from and writing to the stream, try to imagine this analogy. The programming books refer to this mechanism as the **current file position**, and we'll also use this term

An object of an adequate class is created when you open the file and annihilate it at the time of closing. Between these two events, you can use the object to specify what operations should be performed on a particular stream. The operations you're allowed to use are imposed by the way in which you've opened the file. In general, the object comes from one of the classes shown here:

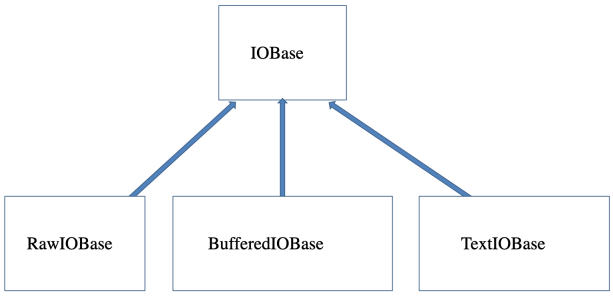

Figure 8: I/O Classes

If you want to get rid of the object, you invoke the method named `close()`. The invocation will sever the connection to the object, and the file and will remove the object.

7.3.1 Reading files

- **Text File**: These ones are structured in lines; that is, they contain typographical characters (letters, digits, punctuation, etc.) arranged in rows (lines), as seen with the naked eye when you look at the contents of the file in the editor. This file is written (or read) mostly **character by character, or line by line.**
- **Others**: The former files don't contain text but a sequence of bytes of any value. This sequence can be, for example, an executable program, an image, an audio or a video clip, a database file, etc. Because these files don't contain lines, the reads and writes relate to portions of data of any size. Hence the data is **read/written byte by byte, or block by block, where the size of the block usually ranges from one to an arbitrarily chosen value.**

7.3.2 End-OF-Line issue

Since portability issues were (and still are) very serious, a decision was made to definitely resolve the issue in a way that doesn't engage the developer's attention.

- when the stream is open and it's advised that the data in the associated file will be processed as text (or there is no such advisory at all), it is switched into text mode;
- during reading/writing of lines from/to the associated file, nothing special occurs in the Unix environment, but when the same operations are performed in the Windows environment, a process called a **translation of newline characters** occurs: when you read a line from the file, every pair of `\r\n` characters is replaced with a single `\n` character, and vice versa; during write operations, every `\n` character is replaced with a pair of `\r\n` characters;
- the mechanism is completely transparent to the program, which can be written as if it was intended for processing Unix/Linux text files only; the source code run in a Windows environment will work properly, too;
- when the stream is open and it's advised to do so, its contents are taken as-is, without any conversion – no bytes are added or omitted.

7.4 Error handling

Fortunately, there is a function that can dramatically simplify the error handling code. It's named `strerror()`, it comes from `os` module and expects just one argument – an error number. Its role is simple: you give an error number and get a string describing the meaning of the error. Note: if you pass a non-existent error code (a number which is not bound to any actual error), the function will raise `ValueError` exception.

```
from os import strerror
try:
        stream = open("path/to/file","rt")
        # actual processing goes here
        stream.close()
except Exception as exc:
        print("File could not be opened:",strerror(exc.errno))
```

7.5 `read()`

If applied to a text file, the function is able to:

- read a desired number of characters (including just one) from the file, and return them as a string;
- read all the file contents, and return them as a string;
- if there is nothing more to read (the virtual reading head reaches the end of the file), the function **returns** an **empty string**.
- `read()` function, invoked without any arguments or with an argument that evaluates to `None`, will read the hole file as once. Pay attention, doing so for a big file could corrupt your OS.

```
from os import strerror
try:
        cnt = 0
        s = open('text.txt',"rt")
        ch = s.read(1)
        while ch != '':
                cnt += 1
                ch = s.read(1)
        s.close()
```

```
        print("Characters in file:", cnt)
except IOError as e:
        print("I/O error occurred: ", strerror(e.errno))
```

Code Result

```
Characters in file: 446
```

7.6 readline()

If you want to treat the file's contents as a set of lines, not a bunch of characters, the **readline()** method will help you with that. The method tries to read a complete line of text from the file, and returns it as a string in the case of success. Otherwise, it returns an **empty string.**

```
from os import strerror
try:
        ccnt = lcnt = 0
        s = open('text.txt','rt')
        line = s.readline()
        while line != '':
                lcnt += 1
                for ch in line:
                        ccnt += 1
                line = s.readline()
        s.close()
        print("Characters in file:", ccnt)
        print("Lines in file:     ", lcnt)
except IOError as e:
        print("I/O error occurred: ", strerr(e.errno))
```

Code Result

```
Characters in file: 446
Lines in file:      7
```

7.7 readlines()

The `readlines()` method, when invoked without arguments, tries to read all the file contents, and **returns a list of strings, one element per file**

line. If you're not sure that the file size is small enough and don't want to test the OS, you can convince the `readlines()` method to read **not more than a specified number of bytes** at once (the returning value remains the same – it's a list of a string). The maximum accepted input buffer size is passed to the method as its argument. Note that when there is nothing to read from the file, the method returns an **empty list**. Use it, for instance it's length to detect the end of the file.

```
from os import strerror
try:
        ccnt = lcnt = 0
        s = open('text.txt','rt')
        lines = s.readlines(20)
        while len(lines) != 0:
                for line in lines:
                        lcnt += 1
                        for ch in line:
                                ccnt += 1
                lines = s.readline(10)
        s.close()
        print("Characters in file:", ccnt)
        print("Lines in file:     ", lcnt)
except IOError as e:
        print("I/O error occurred: ", strerr(e.errno))
```

Code Result

```
Characters in file: 446
Lines in file:      383
```

7.8 `write()`

Writing text files seems to be simpler, as in fact there is one method that can be used to perform such a task. The method is named `write()` and it expects just one argument – a string that will be transferred to an open file (don't forget – the open mode should reflect the way in which the data is transferred – writing a file opened in read mode won't succeed). No newline character is added to the ~write()~'s argument, so you have to add it yourself if you

want the file to be filled with a number of lines. We've decided to write the string's contents character by character (this is done by the inner for loop) but you're not obliged to do it in this way. We just wanted to show you that `write()` is able to operate on single characters.

```
from os import strerror
try:
        fo = open('newtext.txt','wt')
        for i in range(10):
                s = "#" + str(i+1) + " "
                for ch in s:
                        fo.write(ch)
        fo.close()
except IOError as e:
        print("I/O error occurred: ", strerr(e.errno))

try:
    stream = open('newtext.txt', "rt")
    print(stream.read())
except Exception as e:
        print("I/O error eccurred: ", strerr(e.errno))
```

Code Result

```
#1 #2 #3 #4 #5 #6 #7 #8 #9 #10
```

7.8.1 Write to stderr

You can use the same method to write to the `stderr` stream, but don't try to open it, as it's always open implicitly. For example, if you want to send a message string to stderr to distinguish it from normal program output, it may look like this:

```
import sys
sys.stderr.write("Error message")
```

7.9 Amorphous data

Amorphous data is data which have no specific shape or form – they are just a series of bytes. The most important aspect of this is that in the place where we have contact with the data, we are not able to, or simply don't want to, know anything about it. Amorphous data cannot be stored using any of the previously presented means – they are neither strings nor lists. There should be a special container able to handle such data. Python has more than one such container – one of them is a specialized class name `bytearray` – as the name suggests, it's an array containing (amorphous) bytes.

If you want to have such a container, e.g., in order to read in a bitmap image and process it in any way, you need to create it explicitly, using one of available constructors.

```
data = bytearray(100)
```

Such an invocation creates a bytearray object able to store bytes. such a constructor fills the whole array with zeros.

Byte arrays **resemble lists in many respects.** For example, they are mutable, they're a subject of the `len()` function, and you can access any of their elements using **conventional indexing.** There is one important limitation – **you mustn't set any byte array elements with a value which is not an integer** (violating this rule will cause a `TypeError` exception) and you're not allowed to assign a value that doesn't come from the range* 0 to 255* inclusive (unless you want to provoke a `ValueError` exception).

7.9.1 Write bytearray to binary file

Now we're going to show you how to write a byte array to a binary file – binary, as we don't want to save its readable representation – we want to write a **one-to-one** copy of the physical memory content, byte by byte.

```
1 from os import strerror
2
3 data = bytearray(10)
4 for i in range(len(data)):
5         data[i] = 10 + i
6
7 print ("The Binary list: ", data)
```

```
try:
        bf = open('file.bin', 'wb')
        bf.write(data)
        bf.close()
except IOError as e:
        print("I/O error occurred: ", strerr(e.errno))

try:
        sb = open('file.bin', 'rb')
        print("The Binary file: ", sb.read())
except Exception as e:
        print("I/O error occurred:", strerr(e.errno))
```

Code Result

```
The Binary list:  bytearray(b'\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13')
The Binary file:  b'\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13'
```

7.9.2 readinto()

Reading from a binary file requires use of a specialized method name readinto(), as the method doesn't create a new byte array object, but fills a previously created one with the values taken from the binary file.

- the method returns the number of successfully read bytes;
- the method tries to fill the whole space available inside its argument; if there are more data in the file than space in the argument, the read operation will stop before the end of the file; otherwise, the method's result may indicate that the byte array has only been filled fragmentarily (the result will show you that, too, and the part of the array not being used by the newly read contents remains untouched)

```
from os import strerror

data = bytearray(10)
try:
        bf = open('file.bin', 'rb')
        bf.readinto(data)
        bf.close()
        for b in data:
```

```
                print(hex(b), end=' ')
except IOError as e:
        print("I/O error occurred: ", strerr(e.errno))
```

Code Result

```
0xa 0xb 0xc 0xd 0xe 0xf 0x10 0x11 0x12 0x13
```

- An alternative way of reading binary files via `read()` method. If the `read()` method is invoked with an argument, it specifies the maximum number of bytes to be read.

```
from os import strerror
try:
        bf = open('file.bin', 'rb')
        data = bytearray(bf.read())
        bf.close()
        for b in data:
                print(hex(b), end=' ')
except IOError as e:
        print("I/O error occurred: ", strerr(e.errno))
```

Code Result

```
0xa 0xb 0xc 0xd 0xe 0xf 0x10 0x11 0x12 0x13
```

7.10 Copy file program

```
from os import strerror

srcname = input("Source file name?: ")
try:
        src = open(srcname, 'rb')
except IOError as e:
        print("Cannot open source file: ", strerror(e.errno))
        exit(e.errno)
dstname = input("Destination file name?: ")
try:
        dst = open(dstname, 'wb')
except Exception as e:
        print("Cannot create destination file: ", strerr(e.errno))
        src.close()
        exit(e.errno)

buffer = bytearray(65536)
total  = 0
try:
        readin = src.readinto(buffer)
        while readin > 0:
                written = dst.write(buffer[:readin])
                total += written
                readin = src.readinto(buffer)
except IOError as e:
        print("Cannot create destination file: ", strerr(e.errno))
        exit(e.errno)

print(total,'byte(s) succesfully written')
src.close()
dst.close()
```

8 Advanced Bonuses

8.1 Generators

A Python generator is a piece of specialized code able to produce a series of values, and to control the iteration process.

A function returns **one, well-defined value** – it may be the result of a more or less complex evaluation of, e.g., a polynomial, and is **invoked once – only once**. A generator returns a **series of values**, and in general, is (implicitly) **invoked more than once**.

8.2 Iteration Protocol

The iterator protocol is a way in which an object should behave to conform to the rules imposed by the context of the for and in statements. An object conforming to the iterator protocol is called an *iterator*(generator). An iterator must provide two methods:

- `__iter__()` which should return the **object itself** and which is invoked **once** (it's needed for Python to successfully start the iteration)
- `__next__()` which is intended to return the **next value** (first, second, and so on) of the desired series – it will be invoked by the for/in statements in order to pass through the next iteration; if there are no more values to provide, **the method should raise** the `StopIteration` exception.

Sample Code:

```
1  class Fib:
2          def __init__(self, nn):
3                  self.__n = nn
4                  self.__i = 0
5                  self.__p1 = self.__p2 = 1
6
7          def __iter__(self):
8                  print("Fib iter")
9                  return self
10
11         def __next__(self):
```

```
                self.__i += 1
                if self.__i > self.__n:
                        raise StopIteration
                if self.__i in [1,2]:
                        return 1
                ret = self.__p1 + self.__p2
                self.__p1, self.__p2 = self.__p2, ret
                return ret


class Class:
        def __init__(self,n):
                self.__iter = Fib(n)

        def __iter__(self):
                print("Class iter")
                return self.__iter


object = Class(8)

for i in object:
        print(i, end=" ")
```

Code Result

```
Class iter
1 1 2 3 5 8 13 21
```

8.3 yield

Python offers a more effective, convenient, and elegant way of writing iterators. The concept is fundamentally based on a very specific and powerful mechanism provided by the keyword `yield`.

Take a look at this function:

```
def fun(n):
    for i in range(n):
        return i
```

It looks strange, doesn't it? It's clear that the for loop has no chance to finish its first execution, as the return will break it irrevocably. Moreover, invoking the function won't change anything the for loop will start from scratch and will be broken immediately. We can say that such a function is not able to save and restore its state between subsequent invocations. This also means that a function like this *cannot be used as a generator.

Now check this code:

```
def fun(n):
    for i in range(n):
        yield i
```

We've added `yield` instead of `return`. This little amendment turns the function into a **generator**, and executing the `yield` statement has some very interesting effects.

- First of all, it **provides the value of the expression** specified after the `yield` keyword, just like `return`, but doesn't lose the state of the function.
- All the **variables' values are frozen,** and wait for the next invocation, when the execution is **resumed** (not taken from scratch, like after `return`).
- There is one important limitation: such a **function should not be invoked explicitly** as – in fact – it isn't a function anymore; it's a ***generator object.** The invocation will return the object's identifier, not the series we expect from the generator.
- Due to the same reasons, the previous function (the one with the return statement) may only be invoked explicitly, and must not be used as a generator.

Sample code 1:

```
def fun(n):
    for i in range(n):
        yield i

for v in fun(5):
    print(v, end=" ")
```

Code Result

```
0 1 2 3 4
```

Sample code 2:

```
def PowersOf2(n):
        pow = 1
        for i in range(n):
                yield pow
                pow *= 2

for v in PowersOf2(8):
        print(v, end=" ")
```

Code Result

```
1 2 4 8 16 32 64 128
```

Sample code 3: Fibonacci

```
def Fib(n):
        p = pp = 1
        for i in range(n):
                if i in [0,1]:
                        yield 1
                else:
                        n = p + pp
                        pp, p = p, n
                        yield n

fibs = list(Fib(10))
print(fibs)
```

Code Result

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

8.4 Conditional Expression

Conditional expression is a way of selecting one of two different values based on the result of a Boolean expression.

```
expression1 if condition else expression2
```

It may look a bit surprising at first glance, but you have to keep in mind that **it is not a conditional** instruction. Moreover, it's not an instruction at all. It's an **operator**.

Sample Code1:

```
list = []

for x in range(10):
        list.append( 1 if x % 2 == 0 else 0 )

print(list)
```

Code Result

```
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
```

Sample Code2:

```
list = [ 1 if x % 2 == 0 else 0 for x in range(10) ]

print(list)
```

Code Result

```
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
```

8.5 Comprehensions vs Generators

The **brackets** make a comprehension – the parentheses make a **generator.**

```
list = [ 1 if x % 2 == 0 else 0 for x in range(10) ]
genr = ( 1 if x % 2 == 0 else 0 for x in range(10) )

for v in list:
        print(v, end=" ")
print()

for v in genr:
        print(v, end=" ")
print()
```

Code Result

```
1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0
```

There is some proof we can show you. Apply the `len()` function to both these entities. `len(list)` will evaluate to 10. Clear and predictable. `len(genr)` will raise an exception, and you will see the following message:

```
TypeError:  object of type 'generator' has no len()
```

8.6 lambda (anonymous function)

A lambda function is a function without a name (you can also call it an anonymous function). Of course, such a statement immediately raises the question: how do you use anything that cannot be identified? Fortunately, it's not a problem, as you can name such a function if you really need, but, in fact, in many cases the lambda function can exist and work while **remaining fully incognito.**

```
lambda parameters:  expression
```

The declaration of the lambda function doesn't resemble a normal function declaration in any way. Such a clause returns the value of the expression when taking into account the current value of the current lambda argument.

Sample code:

```
two = lambda : 2
sqr = lambda x : x * x
pwr = lambda x,y : x ** y

for a in range(-2,3):
        print(sqr(a), end = ' ')
        print(pwr(a,two()))
```

Code Result

```
4 4
1 1
0 0
1 1
4 4
```

8.7 map()

The **map()** function applies the function passed by its first argument to all its second argument's elements, and **returns an iterator delivering all subsequent function results.** You can use the resulting iterator in a loop, or convert it into a list using the list() function.

```
map(function, list)
```

Sample Code:

```
list1 = [ x for x in range(5) ]
list2 = list(map(lambda x: 2 ** x, list1))
print(list2)
for x in map(lambda x: x * x, list2):
        print(x, end=' ')
print()
```

Code Result

```
[1, 2, 4, 8, 16]
1 4 16 64 256
```

8.8 filter()

Another Python function which can be significantly beautified by the application of a lambda is **filter()**. It expects the same kind of arguments as **map()**, but does something different – it filters its second argument while being guided by directions flowing from the function specified as the first argument (the function is invoked for each list element, just like in **map()**). The elements which return True from the function pass the filter – the others are rejected.

Sample Code:

```
from random import seed, randint

seed()
data = [ randint(-10,10) for x in range(5) ]
filtered = list(filter(lambda x: x > 0 and x % 2 == 0, data))
print(data)
print(filtered)
```

Code Result

```
[3, -9, -3, 8, 9]
[8]
```

8.9 closures

There is a brand new element in it – a function (named **inner**) inside another function (named **outer**).

```
def outer(par):
        loc = par
        def inner():
                return loc
        return inner

var = 1
fun = outer(var)
print(fun())
```

Code Result

```
1
```

- the `inner()` function returns the value of the variable accessible inside its scope, as `inner()` can use any of the entities at the disposal of `outer()`
- the `outer()` function returns the `inner()` function itself; more precisely, it returns a copy of the `inner()` function, the one which was frozen at the moment of ~outer()~'s invocation; the frozen function contains its **full environment,** including the state of all local variables, which also means that the value of loc is successfully retained, although `outer()` ceased to exist a long time ago.
- **The function returned during the outer() invocation is a closure.**
- A closure has to be invoked in exactly the same way in which it has been declared.

```
1  def makeclosure(par):
2          loc = par
3          def power(p):
4                  return p ** loc
5          return power
6
7  fsqr = makeclosure(2)
8  fcub = makeclosure(3)
9  for i in range(5):
10         print(i,fsqr(i),fcub(i))
```

Code Result

```
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
```