Advanced Programming with Python

DISCLAIMER: The presented material relies heavily on Python Advance course carried out at CERN. The material is also available freely at the website: [https://www.python-course.eu (https://www.python-course.eu)](https://www.python-course.eu)

1. What is a variable
2. Basic types
   - string
   - enum
3. Containers
   - lists
   - tuples
   - sets
   - dictionaries
4. Functions
   - arguments
   - recursion
   - static variables
   - decorators
   - generators
   - context managers
5. Exception Handling
   Not included:
6. Object Oriented Programming
7. Packaging
8. Documentation
9. Unit testing
10. Continuous Integration

In 1999, Guido Van Rossum submitted a funding proposal to DARPA called "Computer Programming for Everybody", in which he further defined his goals for Python:

- An easy and intuitive language just as powerful as major competitors
- Open source, so anyone can contribute to its development
- Code that is as understandable as plain English
- Suitability for everyday tasks, allowing for short development times

0. Hello world

In [1]:

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

```
Hello world!
```

0.1. Zen of Python

In [2]:

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1. What is a variable?

Variable in python is always a reference to an object as in python everything, even a function, is an object.

In [3]:

```
x = 3
y = x
y, x
```

Out[3]:

```
(3, 3)
```

In [4]:

```
x = 2
```

In [5]:

```
y, x
```

Out[5]:

```
(3, 2)
```

Conditional statement to assign a value

In [6]:

```
x = -5
if x > 0:
    label = 'Pos'
else:
    label = 'Neg'
print(label)
```

```
Neg
```

In [7]:

```
x = -5
label = 'Pos' if x > 0 else 'Neg'
print(label)
```

```
Neg
```

In [28]:

```
print('Pos' if x > 0 else 'Neg')
```

```
Neg
```

2. Basic types

2.1. String

Strings in python are immutable

In [14]:

```
string = 'My string'
string[0] = 'T'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-9c1867d9b2ff> in <module>
      1 string = 'My string'
----> 2 string[0] = 'T'

TypeError: 'str' object does not support item assignment
```

In [15]:

```
string.replace('M', 'T')
```

Out[15]:

```
'Ty string'
```

In [16]:

```
string
```

Out[16]:

```
'My string'
```

String is iterable

In [17]:

```
for s in 'My string':
    print(s)
```

```
M
y

s
t
r
i
n
g
```

Formating of strings

In [18]:

```
from datetime import date
'Today is ' + str(date.today()) + '.'
```

Out[18]:

```
'Today is 2019-11-28.'
```

In [23]:

```
'Today is {} and number {}.'.format(date.today(), [1, 2, 3])
```

Out[23]:

```
'Today is 2019-11-28 and number [1, 2, 3].'
```

f-strings have been introduced in Python 3.6

In [21]:

```
print(f'Today is {date.today()}')
```

```
Today is 2019-11-28
```

Check if a substring is in a string

In [25]:

```
if 'sub' in 'substring':
    print('True')
```

```
True
```

There are already many built-in functions for handling strings in Python

In [29]:

```
dir(list)
```

Out[29]:

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

In [26]:

```
dir(str)
```

Out[26]:

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
```

```
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

In [32]:

```
'my first sentence'.upper()
```

Out[32]:

```
'MY FIRST SENTENCE'
```

2.2. Enum

Enum is a data type which links a name to an index. They are useful to represent a closed set of options

In [33]:

```
from enum import Enum

class QhBrowserAction(Enum):
    QUERY_BUTTON_CLICKED = 1
    SAVE_BUTTON_CLICKED = 2
    DATE_CHANGED = 3
    QH_NAME_CHANGED = 4
    SLIDER_MOVED = 5

a = QhBrowserAction.DATE_CHANGED
a.name, a.value
```

Out[33]:

```
('DATE_CHANGED', 3)
```

In [36]:

```
a_next = QhBrowserAction(a.value+1)
a_next
```

Out[36]:

```
<QhBrowserAction.QH_NAME_CHANGED: 4>
```

In [38]:

```
if a_next == QhBrowserAction.QH_NAME_CHANGED:
    print('In state {}'.format(a_next.value))
```

```
In state 4
```

3. Containers

Container data types in Python are dedicated to store multiple variables of a various type. The basic container types are: lists, tuples, sets, dictionaries.

3.1. Lists

In [39]:

```
my_list = [1, 'b', True]
my_list
```

Out[39]:

```
[1, 'b', True]
```

Lists are 0-indexed and elements are accessed by a square bracket

In [40]:

```
my_list[0]
```

Out[40]:

```
1
```

Lists are mutable

In [42]:

```
my_list[1] = 0
my_list
```

Out[42]:

```
[0, 0, True]
```

In order to extend a list one can either append...

In [44]:

```
my_list.append(3)
my_list
```

Out[44]:

```
[0, 0, True, 3, 3]
```

Or simply

In [45]:

```
my_list + [1, 'b']
```

Out[45]:

```
[0, 0, True, 3, 3, 1, 'b']
```

...or append elements

In [ ]:

```
my_list += [3]
my_list
```

In [ ]:

```
my_list = my_list + [3] # One shall not do that
my_list
```

Be careful with the last assignment, this creates a new list, so a need to perfom a copy - very inefficient for large lists.

How to append a list at the end?

In [47]:

```
my_list.append([1, 'a'])
my_list
```

Out[47]:

```
[0, 0, True, 3, 3, 3, [1, 'a']]
```

This adds a list as an element, which is not quite what we wanted.

In [58]:

```
my_list.extend([5])
my_list
```

Out[58]:

```
[0, 0, True, 3, 3, 3, [1, 'a'], 1, 'a', 1, 'a', [1, 2], '5', 5]
```

In [53]:

```
import itertools
list2d = [[1,2,3], [4,5,6], [7], [8,9]]
merged = list(itertools.chain(*list2d))
merged
```

Out[53]:

```
[[1, 2, 3], [4, 5, 6], [7], [8, 9]]
```

Which one to choose in order to add elements efficiently?

[https://stackoverflow.com/questions/252703/what-is-the-difference-between-pythons-list-methods-append-and-extend (https://stackoverflow.com/questions/252703/what-is-the-difference-between-pythons-list-methods-append-and-extend)](https://stackoverflow.com/questions/252703/what-is-the-difference-between-pythons-list-methods-append-and-extend)

3.1.1. List comprehension

Old-fashioned way

In [59]:

```
my_list = []
for i in range(10):
    my_list.append(i)
my_list
```

Out[59]:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

One-line list comprehension

In [75]:

```
abs(0.1 - (1.1-1)) < 1e-16
```

Out[75]:

```
True
```

In [65]:

```
my_list = [1/(i+1) for i in range(10)]
my_list
```

Out[65]:

```
[1.0,
 0.5,
 0.3333333333333333,
 0.25,
 0.2,
 0.16666666666666666,
 0.14285714285714285,
 0.125,
 0.1111111111111111,
 0.1]
```

In [66]:

```
my_list = [i for i in range(10) if i > 4]
my_list
```

Out[66]:

```
[5, 6, 7, 8, 9]
```

Generator comprehension

In [76]:

```
x = (x**2 for x in range(10))
print(x)
```

```
<generator object <genexpr> at 0x7faceb983468>
```

In [87]:

```
next(x)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-87-92de4e9f6b1e> in <module>
----> 1 next(x)

StopIteration:
```

In [93]:

```
import datetime
str(datetime.datetime.now())
```

Out[93]:

```
'2019-11-28 11:24:28.029777'
```

In [103]:

```
print(datetime.datetime.now())
for x in ((x+1)**2 for x in range(int(1e7))):
    x**(-1/2)
print(datetime.datetime.now())
```

```
2019-11-28 11:27:55.759043
2019-11-28 11:28:01.770323
```

In [104]:

```
print(datetime.datetime.now())
lst = [(x+1)**2 for x in range(int(1e7))]
for x in lst:
    x**(-1/2)
print(datetime.datetime.now())
```

```
2019-11-28 11:28:09.839305
2019-11-28 11:28:15.530292
```

Generator returns values on demand - no need to create a table and than iterate over it

In [111]:

```
x = iter(range(10))
next(x)
```

Out[111]:

```
0
```

In [ ]:

```
x = (x**2 for x in range(10))
list(x)
```

3.1.2. Filter, map, reduce

In [105]:

```
my_list = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
filter(lambda x: x>0, my_list)
```

Out[105]:

```
<filter at 0x7face70b88d0>
```

Filter returns an iterable generator. Generator is a very important concept in Python!

In [106]:

```
for el in filter(lambda x: x>0,my_list):
    print(el)
```

```
1
2
3
4
5
```

In [112]:

```
list(filter(lambda x: x>0, my_list))
```

Out[112]:

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

Map

In [113]:

```
print(my_list)
list(map(lambda x: abs(x), my_list))
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
```

Out[113]:

```
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5]
```

Map can be applied to many lists

In [114]:

```
lst1 = [0,1,2,3,4]
lst2 = [5,6,7,8]
list(map(lambda x, y: x+y, lst1, lst2))
```

Out[114]:

```
[5, 7, 9, 11]
```

Reduce

In [115]:

```
sum([0,1,2,3,4,5,6,7,8,9,10])
```

Out[115]:

```
55
```

In [116]:

```
from functools import reduce
reduce(lambda x, y: x+y, [0,1,2,3,4,5,6,7,8,9,10])
```

Out[116]:

```
55
```

$0+1+...+n = \frac{n(n+1)}{2}$

3.1.3. Iterating over lists

In [119]:

```
i = 0
for el in [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]:
    print(i, el)
    i += 1
```

```
0 -5
1 -4
2 -3
3 -2
4 -1
5 0
6 1
7 2
8 3
9 4
10 5
```

Iterating with index

In [118]:

```
for index, el in enumerate([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]):
    print(index, el)
```

```
0 -5
1 -4
2 -3
3 -2
4 -1
5 0
6 1
7 2
8 3
9 4
10 5
```

Iterating over two (many) lists

In [120]:

```
letters = ['a', 'b', 'c', 'd']
numbers = [1, 2, 3, 4, 5]
for l, n in zip(letters, numbers):
    print(l, n)
```

```
a 1
b 2
c 3
d 4
```

In [122]:

```
list(zip(letters, numbers))
```

Out[122]:

```
[('a', 1), ('b', 2), ('c', 3), ('d', 4)]
```

In [124]:

```
dict(zip(letters, numbers))
```

Out[124]:

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

In [125]:

```
help(zip)
```

```
Help on class zip in module builtins:

class zip(object)
 |  zip(iter1 [,iter2 [...]]) --> zip object
 |
 |  Return a zip object whose .__next__() method returns a tuple where
 |  the i-th element comes from the i-th iterable argument.  The .__next__()
 |  method continues until the shortest iterable in the argument sequence
 |  is exhausted and then it raises StopIteration.
 |
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __reduce__(...)
 |      Return state information for pickling.
```

3.1.4. Copying lists

In [126]:

```
x = [1, 2, 3, 4]
y = x
y[0] = 'a'
print(x, y)
```

```
['a', 2, 3, 4] ['a', 2, 3, 4]
```

In [128]:

```
x.copy()
```

Out[128]:

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

In [127]:

```
x = [1, 2, 3, 4]
y = x.copy()
y[0] = 'a'
print(x, y)
```

```
[1, 2, 3, 4] ['a', 2, 3, 4]
```

In [129]:

```
x = [[1, 'a'], 2, 3, 4]
y = x.copy() # equivalent to x[:]
y[0] = 'a'
print(x, y)
```

```
[[1, 'a'], 2, 3, 4] ['a', 2, 3, 4]
```

In [131]:

```
x = [[1, 'a'], 2, 3, 4]
y = x.copy()
y[0][0] = 'b'
print(x, y)
```

```
[['b', 'a'], 2, 3, 4] [['b', 'a'], 2, 3, 4]
```

The reason for this behavior is that Python performs a shallow copy.

In [132]:

```
from copy import deepcopy
x = [[1, 'a'], 2, 3, 4]
y = deepcopy(x)
y[0][0] = 'b'
print(x, y)
```

```
[[1, 'a'], 2, 3, 4] [['b', 'a'], 2, 3, 4]
```

3.1.5. Sorting lists - inplace operations

In [133]:

```
x = [1, 10, 2, 9, 3, 8, 4, 6, 5]
x = x.sort()
print(x)
```

```
None
```

list.sort() is an inplace operation. In general, inplace operations are efficient as they do not create a new copy in memory

In [134]:

```
x = [1, 10, 2, 9, 3, 8, 4, 6, 5]
x.sort()
print(x)
```

```
[1, 2, 3, 4, 5, 6, 8, 9, 10]
```

list.sorted does create a new variable

In [135]:

```
x = [1, 10, 2, 9, 3, 8, 4, 6, 5]
sorted(x)
print(x)
```

```
[1, 10, 2, 9, 3, 8, 4, 6, 5]
```

In [136]:

```
x = [1, 10, 2, 9, 3, 8, 4, 6, 5]
x = sorted(x)
print(x)
```

```
[1, 2, 3, 4, 5, 6, 8, 9, 10]
```

In [137]:

```
x = [1, 10, 2, 9, 3, 8, 4, 6, 5]
x is sorted(x)
```

Out[137]:

```
False
```

How to sort in a reverted order

In [139]:

```
x = [1, 10, 2, 9, 3, 8, 4, 6, 5]
x.sort(reverse=True)
print(x)
```

```
[10, 9, 8, 6, 5, 4, 3, 2, 1]
```

Sort nested lists

In [140]:

```
employees = [(111, 'John'), (123, 'Emily'), (232, 'David'), (100, 'Mark'), (1, 'Andrew')]
employees.sort(key=lambda x: x[0])
employees
```

Out[140]:

```
[(1, 'Andrew'), (100, 'Mark'), (111, 'John'), (123, 'Emily'), (232, 'David')]
```

In [141]:

```
employees = [(111, 'John'), (123, 'Emily'), (232, 'David'), (100, 'Mark'), (1, 'Andrew')]
employees.sort(key=lambda x: x[1])
employees
```

Out[141]:

```
[(1, 'Andrew'), (232, 'David'), (123, 'Emily'), (111, 'John'), (100, 'Mark')]
```

Also with reversed order

In [142]:

```
employees = [(111, 'John'), (123, 'Emily'), (232, 'David'), (100, 'Mark'), (1, 'Andrew')]
employees.sort(key=lambda x: x[0], reverse=True)
employees
```

Out[142]:

```
[(232, 'David'), (123, 'Emily'), (111, 'John'), (100, 'Mark'), (1, 'Andrew')]
```

3.1.6. List extras

In [143]:

```
my_list = 5*['a']
my_list
```

Out[143]:

```
['a', 'a', 'a', 'a', 'a']
```

In [144]:

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

Out[144]:

```
True
```

In [149]:

```
x = ['a']
y = ['a']
x == y
```

Out[149]:

```
True
```

In [150]:

```
x = ('a')
y = ('a')
x is y
```

Out[150]:

```
True
```

3.2. Tuples

Tuples, similarly to lists can stores elements of different types.

In [152]:

```
my_tuple = (1,2,3)
my_tuple
```

Out[152]:

```
(1, 2, 3)
```

In [153]:

```
my_tuple[0]
```

Out[153]:

```
1
```

Unlike the lists, tuples are immutable.

In [154]:

```
my_tuple[0]=0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-154-a0c25be542d6> in <module>
----> 1 my_tuple[0]=0

TypeError: 'tuple' object does not support item assignment
```

In [159]:

```
tuple([1,2,3])
```

Out[159]:

```
(1, 2, 3)
```

3.3. Sets

Sets are immutable and contain only unique elements

In [155]:

```
{1,2,3,4}
```

Out[155]:

```
{1, 2, 3, 4}
```

In [156]:

```
{1,2,3,4,4}
```

Out[156]:

```
{1, 2, 3, 4}
```

So this is a neat way for obtaining unique elements in a list

In [157]:

```
my_list = [1, 2, 3, 4, 4, 5, 5, 5]
set(my_list)
```

Out[157]:

```
{1, 2, 3, 4, 5}
```

or a tuple

In [158]:

```
my_tuple = (1, 2, 3, 4, 4, 5, 5, 5)
set(my_tuple)
```

Out[158]:

```
{1, 2, 3, 4, 5}
```

One can perform set operations on sets ;-)

In [160]:

```
A = {1,2,3}
B = {3,4,5}
print(f'A+B={A.union(B)}')
print(f'A-B={A-B}')
print(f'A*B={A.intersection(B)}')
print(f'A*0={A.intersection({})}')
```

```
A+B={1, 2, 3, 4, 5}
A-B={1, 2}
A*B={3}
A*0=set()
```

In [165]:

```
pm = {'system', 'source', 'I_MEAS', 'I_REF'}
signals = pm - {'system', 'source'}
signals
```

Out[165]:

```
{'I_MEAS', 'I_REF'}
```

In [174]:

```
for s in signals:
    print(s)
```

```
I_MEAS
I_REF
```

In [175]:

```
help(set)
```

```
Help on class set in module builtins:

class set(object)
 |  set() -> new empty set object
 |  set(iterable) -> new set object
 |
 |  Build an unordered collection of unique elements.
 |
 |  Methods defined here:
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iand__(self, value, /)
 |      Return self&=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __ior__(self, value, /)
 |      Return self|=value.
 |
 |  __isub__(self, value, /)
 |      Return self-=value.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __ixor__(self, value, /)
 |      Return self^=value.
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
```

```
 |  
 |  __lt__(self, value, /)
 |      Return self<value.
 |  
 |  __ne__(self, value, /)
 |      Return self!=value.
 |  
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |  
 |  __or__(self, value, /)
 |      Return self|value.
 |  
 |  __rand__(self, value, /)
 |      Return value&self.
 |  
 |  __reduce__(...)
 |      Return state information for pickling.
 |  
 |  __repr__(self, /)
 |      Return repr(self).
 |  
 |  __ror__(self, value, /)
 |      Return value|self.
 |  
 |  __rsub__(self, value, /)
 |      Return value-self.
 |  
 |  __rxor__(self, value, /)
 |      Return value^self.
 |  
 |  __sizeof__(...)
 |      S.__sizeof__() -> size of S in memory, in bytes
 |  
 |  __sub__(self, value, /)
 |      Return self-value.
 |  
 |  __xor__(self, value, /)
 |      Return self^value.
 |  
 |  add(...)
 |      Add an element to a set.
 |      
 |      This has no effect if the element is already present.
 |  
 |  clear(...)
 |      Remove all elements from this set.
 |  
 |  copy(...)
 |      Return a shallow copy of a set.
 |  
 |  difference(...)
 |      Return the difference of two or more sets as a new set.
 |      
 |      (i.e. all elements that are in this set but not the others.)
 |  
 |  difference_update(...)
 |      Remove all elements of another set from this set.
 |  
 |  discard(...)
 |      Remove an element from a set if it is a member.
 |      
 |      If the element is not a member, do nothing.
 |  
 |  intersection(...)
 |      Return the intersection of two sets as a new set.
 |      
 |      (i.e. all elements that are in both sets.)
 |  
 |  intersection_update(...)
 |      Update a set with the intersection of itself and another.
 |  
```

```
 |  isdisjoint(...)
 |      Return True if two sets have a null intersection.
 |
 |  issubset(...)
 |      Report whether another set contains this set.
 |
 |  issuperset(...)
 |      Report whether this set contains another set.
 |
 |  pop(...)
 |      Remove and return an arbitrary set element.
 |      Raises KeyError if the set is empty.
 |
 |  remove(...)
 |      Remove an element from a set; it must be a member.
 |
 |      If the element is not a member, raise a KeyError.
 |
 |  symmetric_difference(...)
 |      Return the symmetric difference of two sets as a new set.
 |
 |      (i.e. all elements that are in exactly one of the sets.)
 |
 |  symmetric_difference_update(...)
 |      Update a set with the symmetric difference of itself and another.
 |
 |  union(...)
 |      Return the union of sets as a new set.
 |
 |      (i.e. all elements that are in either set.)
 |
 |  update(...)
 |      Update a set with the union of itself and others.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

In [177]:

```
signals[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-177-6c9ebb69209b> in <module>
----> 1 signals[0]

TypeError: 'set' object does not support indexing
```

In [180]:

```
next(iter(signals))
```

Out[180]:

```
'I_MEAS'
```

In [173]:

```
list(signals)[0]
```

Out[173]:

```
'I_MEAS'
```

Unpacking variables

In [182]:

```
first, second = [1, 2]
print(first, second)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-182-07dd77cb2d66> in <module>
----> 1 first, second = [1, 2, 3]
      2 print(first, second)

ValueError: too many values to unpack (expected 2)
```

In [183]:

```
first, second = (1, 2)
print(first, second)
```

```
1 2
```

In [184]:

```
first, second = {1, 2}
print(first, second)
```

```
1 2
```

In [185]:

```
employees = [(111, 'John'), (123, 'Emily'), (232, 'David'), (100, 'Mark'), (1, 'Andrew')]
for employee_id, employee_name in employees:
    print(employee_id, employee_name)
```

```
111 John
123 Emily
232 David
100 Mark
1 Andrew
```

3.4. Dictionaries

In [186]:

```
empty_set = {}
type(empty_set)
```

Out[186]:

```
dict
```

In [187]:

```
empty_set = set()
type(empty_set)
```

Out[187]:

```
set
```

In [188]:

```
my_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
my_dict
```

Out[188]:

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

In [189]:

```
my_dict['a']
```

Out[189]:

```
1
```

In [190]:

```
for key in my_dict:
    print(key)
```

```
a
b
c
d
```

In [191]:

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

```
a 1
b 2
c 3
d 4
```

Summary of Python Containers

Feature	list	tuple	dict	set
Purpose	an ordered collection of variables	an ordered collection of variables	an ordered collection of key,value pairs	a collection of variables
Duplication of values	yes	yes	unique keys, duplicate values	no
Mutability	yes	no	yes	no
Creation	[1,2,3]	(1,2,3)	{'a':1}	{1,2,3}
Empty container	[]	()	{}	set()
Comprehension	[x for x in range(5)]	tuple((x for x in range(5)))	{k: v for k,v in zip(['a'], [1])}	{x for x in range(5)}
Accessing element	lst[0]	tpl[0]	dct['key']	not possible

4. Functions

In [1]:

```
# lambda functions
f = lambda x: x**2
f(2)
```

Out[1]:

```
4
```

In [31]:

```
def f(x):
    return x**2
f(2)
```

Out[31]:

```
4
```

4.1. Arguments

In [2]:

```
def f(a, b, c=3):
    return a+b+c

f(1,2)
```

Out[2]:

```
6
```

In [3]:

```
f(1,2, 4)
```

Out[3]:

```
7
```

If the number of arguments matches, one can pass a list

In [5]:

```
lst = [1,2,3]
f(*lst)
```

Out[5]:

```
6
```

or a dictionary (provided that key names match the argument names) - very useful for methods with multiple arguments, e.g., plotting, querying databases, etc.

In [6]:

```
dct = {'a': 1, 'b': 2, 'c': 3}
f(**dct)
```

Out[6]:

```
6
```

In [ ]:

```
query_params = {'db': "NXCALS", "signal": "I_MEAS", "t_start": "today", "t_end": "tomorrow"}
call_db(**query_params)
query_params['db'] = 'PM'
call_db(**query_params)
```

Default argument values

In [8]:

```
def f(a, b, d, c=3):
    return a+b+c+d
```

In [15]:

```
def f(*args):
    print(len(args))
    return args[0]*args[1]*args[2]
f(1, 10, 'a')
```

```
3
```

Out[15]:

```
'aaaaaaaaaa'
```

In [38]:

```
def f(**kwargs):
    return kwargs['a'] + kwargs['b']
f(a=1, b=2, c=3)
```

Out[38]:

```
3
```

In [17]:

```
def f(arg, *args, **kwargs):
    return arg + sum(args) + kwargs['f']

f(1, 2, 3, 4, 5, f=6)
```

Out[17]:

```
21
```

In [18]:

```
def f(a, b, *, c):
    return a+b+c
f(1,2,3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-ff89bb262ade> in <module>
      1 def f(a, b, *, c):
      2     return a+b+c
----> 3 f(1,2,3)

TypeError: f() takes 2 positional arguments but 3 were given
```

In [19]:

```
f(1,2,scaling=3)
```

Out[19]:

```
6
```

A function passed as an argument

In [20]:

```
def f(x):
    return x**2

def g(func, x):
    return func(x)

g(f,2)
```

Out[20]:

```
4
```

A function can return multiple values, in fact it returns a tuple

In [23]:

```
def f():
    return 'a', 'b', 's'
f()
```

Out[23]:

```
('a', 'b', 's')
```

In [32]:

```
first = list(f())
# print(first)
# print(second)
```

In [33]:

```
first[1] = 2
```

In [34]:

```
first
```

Out[34]:

```
['a', 2, 's']
```

4.2. Recursion

Factorial of an integer n is given as: \begin{equation} n! = n*(n-1)*(n-2)*(n-3)* ... * 3 * 2 * 1 \end{equation} For example: \begin{equation} 5! = 5 * 4 * 3 * 2 * 1 = 120 \end{equation}

In [37]:

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)

factorial(3)
```

Out[37]:

```
6
```

In [38]:

```
factorial(5)
```

Out[38]:

```
120
```

In [39]:

```
factorial(-1)
```

```
ERROR:root:Internal Python error in the inspect module.
Below is the traceback from this internal error.
```

```
Traceback (most recent call last):
  File "/usr/local/lib/swan/IPython/core/interactiveshell.py", line 3326, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-39-5aae425d6a8b>", line 1, in <module>
    factorial(-1)
  File "<ipython-input-37-0017e71e028e>", line 5, in factorial
    return n*factorial(n-1)
  File "<ipython-input-37-0017e71e028e>", line 5, in factorial
    return n*factorial(n-1)
  File "<ipython-input-37-0017e71e028e>", line 5, in factorial
    return n*factorial(n-1)
  [Previous line repeated 2967 more times]
  File "<ipython-input-37-0017e71e028e>", line 2, in factorial
    if n == 1:
RecursionError: maximum recursion depth exceeded in comparison

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/usr/local/lib/swan/IPython/core/interactiveshell.py", line 2040, in showtraceback
    stb = value._render_traceback_()
AttributeError: 'RecursionError' object has no attribute '_render_traceback_'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/cvmfs/sft.cern.ch/lcg/releases/Python/3.6.5-f74f0/x86_64-centos7-gcc8-opt/lib/python3.6/genericpath.p
y", line 19, in exists
    os.stat(path)
FileNotFoundError: [Errno 2] No such file or directory: '<ipython-input-37-0017e71e028e>'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/usr/local/lib/swan/IPython/core/ultratb.py", line 1101, in get_records
    return _fixed_getinnerframes(etb, number_of_lines_of_context, tb_offset)
  File "/usr/local/lib/swan/IPython/core/ultratb.py", line 319, in wrapped
    return f(*args, **kwargs)
  File "/usr/local/lib/swan/IPython/core/ultratb.py", line 353, in _fixed_getinnerframes
    records = fix_frame_records_filenames(inspect.getinnerframes(etb, context))
  File "/cvmfs/sft.cern.ch/lcg/releases/Python/3.6.5-f74f0/x86_64-centos7-gcc8-opt/lib/python3.6/inspect.py",
line 1483, in getinnerframes
    frameinfo = (tb.tb_frame,) + getframeinfo(tb, context)
  File "/cvmfs/sft.cern.ch/lcg/releases/Python/3.6.5-f74f0/x86_64-centos7-gcc8-opt/lib/python3.6/inspect.py",
line 1441, in getframeinfo
    filename = getsourcefile(frame) or getfile(frame)
  File "/cvmfs/sft.cern.ch/lcg/releases/Python/3.6.5-f74f0/x86_64-centos7-gcc8-opt/lib/python3.6/inspect.py",
line 693, in getsourcefile
    if os.path.exists(filename):
  File "/cvmfs/sft.cern.ch/lcg/releases/Python/3.6.5-f74f0/x86_64-centos7-gcc8-opt/lib/python3.6/genericpath.p
y", line 19, in exists
    os.stat(path)
KeyboardInterrupt
---------------------------------------------------------------------------
```

In [42]:

```
def factorial(n):
    if not isinstance(n, int) or n <= 0:
        raise ValueError("Argument is not a positive integer")

    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
factorial(5)
```

Out[42]:

```
120
```

Flattening a nested list

In [50]:

```
def flatten_nested_lists(x):
    result = []
    for el in x:
        if isinstance(el, (list, tuple)):
            result.extend(flatten_nested_lists(el))
        else:
            result.append(el)
    return result
```

In [51]:

```
lst1 = [1]
lst2 = [1, 2]
lst1.append(lst2)
lst1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-51-d821110e9bc8> in <module>
      1 lst1 = [1]
      2 lst2 = [1, 2]
----> 3 lst1.append(*lst2)
      4 lst1

TypeError: append() takes exactly one argument (2 given)
```

In [51]:

```
lst = [1, 2, [3,4], [5, [6, 7]]]
flatten_nested_lists(lst)
```

Out[51]:

```
[1, 2, 3, 4, 5, 6, 7]
```

Fibonacci

In [52]:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
[fib(i) for i in range(6)]
```

Out[52]:

```
[0, 1, 1, 2, 3, 5]
```

How many times do we calculate fib(3)?

In [56]:

```
arguments = []
def fib(n):
    arguments.append(n)
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
x = [fib(i) for i in range(6)]
print(x)
```

```
[0, 1, 1, 2, 3, 5]
```

In [58]:

```
counts = {i: arguments.count(i) for i in range(max(arguments)+1)}
counts
```

Out[58]:

```
{0: 8, 1: 12, 2: 7, 3: 4, 4: 2, 5: 1}
```

In [59]:

```
sum(counts.values())
```

Out[59]:

```
34
```

4.3. Memoization

In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

source: [https://en.wikipedia.org/wiki/Memoization (https://en.wikipedia.org/wiki/Memoization)](https://en.wikipedia.org/wiki/Memoization)

In [60]:

```
# Memoization for Fibonacci
# Fibonacci
memo = {0:0, 1:1}
arguments = []
def fib(n):
    arguments.append(n)
    if n not in memo:
        memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
[fib(i) for i in range(6)]
```

Out[60]:

```
[0, 1, 1, 2, 3, 5]
```

In [63]:

```
counts = {i: arguments.count(i) for i in range(max(arguments)+1)}
counts
```

Out[63]:

```
{0: 2, 1: 3, 2: 3, 3: 3, 4: 2, 5: 1}
```

In [62]:

```
sum(counts.values())
```

Out[62]:

```
14
```

4.5. Decorators

Decorators are functions dedicated to enhance functionality of a given function, e.g., check parameter inputs, format input

In [59]:

```
def argument_test_natural_number(f):
    def helper(x):
        if type(x) is int and x > 0:
            return f(x)
        else:
            raise Exception("Argument is not an integer")
    return helper

def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)

factorial = argument_test_natural_number(factorial)
factorial(3)
```

Out[59]:

```
6
```

In [60]:

```
factorial(-1)
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-60-5aae425d6a8b> in <module>
----> 1 factorial(-1)

<ipython-input-59-74d7cacc5284> in helper(x)
      4             return f(x)
      5         else:
----> 6             raise Exception("Argument is not an integer")
      7     return helper
      8

Exception: Argument is not an integer
```

In [64]:

```
def argument_test_natural_number(f):
    def helper(x):
        if type(x) is int and x > 0:
            return f(x)
        else:
            raise Exception("Argument is not an integer")
    return helper

@argument_test_natural_number
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)

factorial(3)
```

Out[64]:

```
6
```

In [65]:

```
factorial(-1)
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-65-5aae425d6a8b> in <module>
----> 1 factorial(-1)

<ipython-input-64-61c7137e6453> in helper(x)
      4             return f(x)
      5         else:
----> 6             raise Exception("Argument is not an integer")
      7     return helper
      8

Exception: Argument is not an integer
```

In [66]:

```
def sum_aritmetic_series(n):
    return n*(n+1)/2
sum_aritmetic_series(2)
```

Out[66]:

```
3.0
```

In [67]:

```
sum_aritmetic_series(1.5)
```

Out[67]:

```
1.875
```

In [68]:

```
@argument_test_natural_number
def sum_aritmetic_series(n):
    return n*(n-1)/2
sum_aritmetic_series(2)
```

Out[68]:

```
1.0
```

In [69]:

```
sum_aritmetic_series(1.5)
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-69-16581ed0c766> in <module>
----> 1 sum_aritmetic_series(1.5)

<ipython-input-64-61c7137e6453> in helper(x)
      4             return f(x)
      5         else:
----> 6             raise Exception("Argument is not an integer")
      7     return helper
      8 

Exception: Argument is not an integer
```

Fixing the Fibonacci series

In [70]:

```
def memoize(f):
    memo = {}
    def helper(n):
        if n not in memo:
            memo[n] = f(n)
        return memo[n]
    return helper

arguments = []

@memoize
def fib(n):
    arguments.append(n)
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
[fib(i) for i in range(6)]
```

Out[70]:

```
[0, 1, 1, 2, 3, 5]
```

In [71]:

```
counts = {i: arguments.count(i) for i in range(max(arguments)+1)}
counts
```

Out[71]:

```
{0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1}
```

In [72]:

```
sum(counts.values())
```

Out[72]:

```
6
```

There is a built-in cache decorator

In [ ]:

```
# built-in least-recently used cache decorator
import functools
@functools.lru_cache(maxsize=128, typed=False)
def fib(n):
    if n < 2:
        return
    else:
        return fib(n-1) + fib(n-2)
```

4.4. Static variables

In [73]:

```
# Exercise
# - write a decorator counting the number of times a function was called
# - the same but for a varying number of parameters and keyword-arguments

def counter(func):
    # first define function
    def helper(x, *args, **kwargs):
        helper.count += 1
        return func(x, *args, **kwargs) # return function as it is
    # then, define an attribute to be incremented with every call
    # this attribute behaves like a static variable
    # helper exist only after the function definition. Once defined, then we can attach an attribute
    helper.count = 0

    return helper

@counter
def fun(x):
    return x

fun(1)
fun(2)
fun(3)
fun.count
```

Out[73]:

```
3
```

4.6. Generators

In [74]:

```
s = "Python"
itero = iter(s)
itero
# what I write is:
# for char in s:
# what python does:
# for char in iter(s)
# in fact it is a while loop until stop is reached
```

Out[74]:

```
<str_iterator at 0x7f4675fb7f28>
```

In [75]:

```
next(itero)
```

Out[75]:

```
'P'
```

In [76]:

```
next(itero)
```

Out[76]:

```
'y'
```

In [77]:

```
next(itero)
```

Out[77]:

```
't'
```

In [78]:

```
next(itero)
```

Out[78]:

```
'h'
```

In [79]:

```
next(itero)
```

Out[79]:

```
'o'
```

In [80]:

```
next(itero)
```

Out[80]:

```
'n'
```

In [81]:

```
next(itero)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-81-bc7ed7acd9c9> in <module>
----> 1 next(itero)

StopIteration:
```

Own generator

In [87]:

```
def abc_generator():
    yield "a"
    yield "b"
    yield "c"

x = abc_generator() # we call like a function. A function returns an object

for i in x:
    print(i)
```

```
a
b
c
```

In [86]:

```
next(x)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-86-92de4e9f6b1e> in <module>
----> 1 next(x)

StopIteration:
```

In [80]:

```
# print(next(x)) <-- yield "a"
# print(next(x)) <-- yield "b"
# print(next(x)) <-- yield "c"
# this is a co-process. This function creates a code waiting to be executed, when we assign x = abc_generator(
)
# after it reaches a yield, it returns value and stops. Then next is positioned fter the yield.x
x = abc_generator()
print(next(x))
print(next(x))
print(next(x))
print(next(x))
```

```
a
b
c
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-80-e58ce091c0f7> in <module>
      8 print(next(x))
      9 print(next(x))
---> 10 print(next(x))

StopIteration:
```

A function is also a single-value generator

In [88]:

```
def abc_generator():
    return "a"
x = abc_generator()
for i in x:
    print(i)
# works, because the returned value is iterable
```

```
a
```

In [82]:

```
type(abc_generator())
```

Out[82]:

```
str
```

In [83]:

```
def abc_generator():
    for char in ["a", "b", "c"]:
        yield char
for i in abc_generator():
    print(i)
```

```
a
b
c
```

In [85]:

```
type(abc_generator())
```

Out[85]:

```
generator
```

In [ ]:

```
# Generate a pi value
# pi/4 = 1 - 1/3 + 1/5 - 1/7

def pi_series():
    sum = 0
    i = 1.0
    j = 1
    while True:
        sum = sum + j/i
        yield 4*sum
        i = i + 2
        j = j * -1
# runs forever
# we can break with a counter, but it is not a good idea
for i in pi_series():
    print(i)
```

In [ ]:

```
def firstn(g, n):
    for i in range(n):
        yield next(g)

print(list(firstn(pi_series(), 8)))
```

4.7. Context Manager

Is used to allocate and release some sort of resource when we need it.
Which means that before we start a block we open e.g. a file, and when we are going out, the file is automatically released.
If we don't close, it remains open in a file system. Closing a program, it would close. A good practice is to always close.

With context managers, the benefit is no need to close.
The issue is with the exceptions. With with, the exception is caught and handled.
Context manager is a general concept. The concept is as follows.

```
with device():

before:
1. check device
2. start device

we enter the block:
1. we do something

after:
1. we execute stop block

in case of exceptions we are sure that the after part will be executed.
```

In [ ]:

```
import csv
with open('example.txt', 'w') as out:
    csv_out = csv.writer(out)
    csv_out.writerow(['date', '# events'])
```

In [ ]:

```python
from contextlib import contextmanager

@contextmanager
def device():
    print("Check device")
    device_state = True
    print("Start device")
    yield device_state # the block after with is executed
    print("Stop device")

with device() as state:
    print("State is ", state)
    print("Device is running!")
```

5. Exception handling

Exception handling
It is easier to ask for forgiveness than for permission
E.g.

```
if fileexisits(file_name):
   txt = open(file_name).read()
```

We first check if the file exists, then in the next step we fetch the file - two operations (asking for permission)
We can try to read, if it is there we are good, otherwise it raises an exception - single operation (asking for forgiveness)

```
try:
   txt = open(file_name)
except Exception as e:
   txt = ""
```

In [ ]:

```python
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError as err:
        print("Error message: ", err)
        print("No valid number. Try again")
```

```
try:
  some code
except ZeroDivisionError:
  some code
  there could be a raise here
except FooError:
  some code
except BarError:
  some code
finally:
  some code executed always
```

In [ ]:

```
# Finally is executed always
try:
    x = float(input("Your number: "))
    inverse = 10/x
except ValueError as err:
        print("Error message: ", err)
        print("No valid number. Try again")
finally:
    print("There may or may not have been an exception.")
print("The inverse: ", inverse)
```

In [ ]:

```
# assert
x = 5
y = 6
assert x < y, "x has to be smaller than y"
```