02_Data_structures

March 9, 2022

```
[1]: from IPython.display import Image
```

1 Programming Python

2 Used modules

```
[148]: import math
```

3 A quick resume: Why Python?

Let's review some fundamental features of the Python Language:

Designed for **code readability** and teaching purposes. Defines **blocks** with indentation:

read a file in Python:

```
in = open("input.txt","r")
out = open("output.txt", "w")
out.writelines(in)
in.close()
out.close()
```

or

```
open("output.txt", "w").writelines(open("input.txt"))
```

read a file in C:

```
#include <stdio.h>
int main(int argc, char **argv) {
    FILE *in, *out;
    int c;
    in = fopen("input.txt", "r");
    out = fopen("output.txt", "w");
    while ((c = fgetc(in)) != EOF) {
        fputc(c, out);
    }
    fclose(out);
    fclose(in);
}
```

Dynamic typing: the type of a variable is set from its content, the way it is used and the methods which can be applied to it (with restrictions)

Duck Typing: When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck. • The type of a variable is based on its methods and properties and not explicitly provided.

• An important feature of duck typing is that variables with different properties can still be used in the same sequence of instructions provided they have the properties and methods needed in this sequence.

• A consequence of this feature is that procedures can be simply used with arguments of different types if they support the instructions present in the procedure.

Basic types:

- integers
- floats
- complex
- booleans
- strings

strings are *sequences of characters* (also, *we count from 0!*):

[2]:
```
s="mystring"
print(s[0])
s*2
```

```
m
```

[2]:
```
'mystringmystring'
```

Make decisions

```
if a:
    do something
elif b:
    do something else
else:
    yet another option
```

Flow control

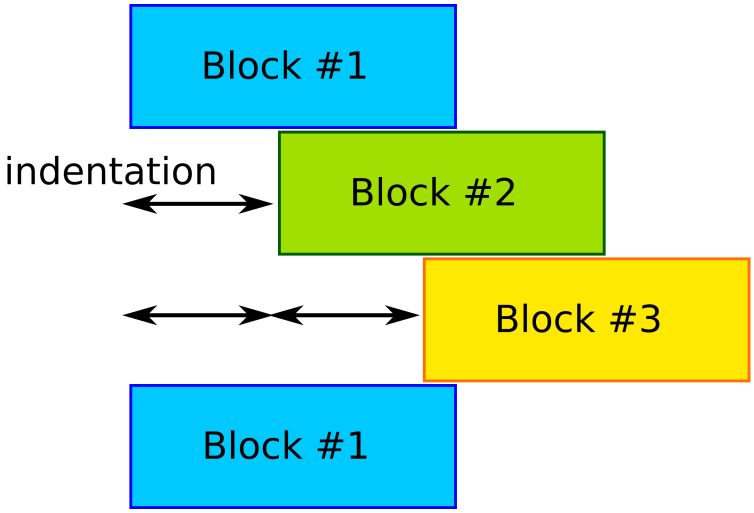
```
for i in range(start,end,step):
    do something
    if a: break
    if b: continue

while condition is True:
    do something
    evaluate condition
```

Blocks are managed with indentation

[3]:
```
Image(filename="blocks.png")
```

[3]:



IPython magic commands (many more available):

%reset: clear all namespace
%hist: print history
%xdel: delete variables
%who: list objects in enviroment (use in combo with ? to know who is who)

3.0.1 Define functions

Functions are created with the **def** statement and return (sequences of) values with **return**

```
[4]: def addition(arg1,arg2):
         return arg1+arg2
     addition(2,2)
```

```
[4]: 4
```

some arguments may be *keywords* with a default value:

```
[5]: def power(arg1,arg2=2):
         return arg1**arg2
     power(3)
```

```
[5]: 9
```

3.1 File Manipulation

To open a file:

open(filename, mode)

mode is one of: `r' (Read), `w' (Write), `a' (Append). If a file opened for `w' does not exist it will be created.

Common methods for file handles include:

f.readline(): read a single line from a file. Returns a string.
f.readlines([size]): read all the lines up to size bytes and return them in a iterator of stri
f.read([size]): read up to size bytes; returns a string
f.write(text): write text to file

```
[6]: f = open('foo.txt')
```

```
       ␣
 ↪---------------------------------------------------------------------------

        FileNotFoundError                          Traceback (most recent call␣
 ↪last)
```

```
    /tmp/ipykernel_233018/1650707477.py in <module>
----> 1 f = open('foo.txt')


    FileNotFoundError: [Errno 2] No such file or directory: 'foo.txt'
```

[7]:

```
f = open('foo.txt',"w")
f.write("Dutch Dillon Billy")
f.close()
```

[8]:

```
!ls -rt
```

```
sequence.py  dict.png       blocks.png        01b_Programming_Python.ipynb
scope.png    babynames3.py  hivsequences.txt  foo.txt
pass.c       baby2006.html  byvalue.png
list.png     baby2008.html  byref.png
```

[9]:

```
!cat foo.txt
```

```
Dutch Dillon Billy
```

[10]:

```
!rm -f foo.txt
```

3.2 Exercise

Newton's method allows to estimate the square root of a number a from an initial estimate x:

$$y = \frac{x + a/x}{2}$$

write a function the implements and uses some convergence criterion.

[11]:

```
#Solution
def newton(a,x,conv=1e-4,maxit=100):
    val = a-x
    for i in range(maxit):
        y = 0.5 * (x+a/x)
        if abs(y-x) <= conv:
            break
        x = y
    return y,i
print(newton(4,3))
newton(81,23)
```

```
(2.0000000000262146, 3)
```

```
[11]: (9.00000000005842, 4)
```

4 Builtins

Some **keywords** are reserved by the interpreter to define the basics constructs of the language (*semantics*):

and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield

stating that reassigning them will trigger a SyntaxError:

```
[14]: finally=10
```

```
  File "/tmp/ipykernel_233018/20273995.py", line 1
    finally=10
    ^
SyntaxError: invalid syntax
```

A complete list of reserved keywords can be printed in the interpreter in this way:

```
[15]: import keyword
      keyword.kwlist
```

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

What about other statements that have been presented such as **sum** or **next**?

These are **built-in** functions that are automatically available without importing any module; they are not keywords however and can be reassigned with something like:

```
[16]: sum([1,2,3])
```

```
[16]: 6
```

```
[17]: sum="sum"
```

```
[18]: sum([1,2,3])
```

```
       ␣
 ↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
 ↪last)

        /tmp/ipykernel_233018/980638477.py in <module>
    ----> 1 sum([1,2,3])


        TypeError: 'str' object is not callable
```

```
[19]: del sum
```

```
[20]: sum([1,2,3])
```

```
[20]: 6
```

later on, read about the %reset Ipython magic

A list of built-in is available at https://docs.python.org/3.8/library/functions.html and can be obtained in this way:

```
[21]: import builtins
      dir(builtins)
```

```
[21]: ['ArithmeticError',
       'AssertionError',
       'AttributeError',
       'BaseException',
       'BlockingIOError',
       'BrokenPipeError',
       'BufferError',
       'BytesWarning',
       'ChildProcessError',
       'ConnectionAbortedError',
       'ConnectionError',
       'ConnectionRefusedError',
       'ConnectionResetError',
       'DeprecationWarning',
       'EOFError',
       'Ellipsis',
       'EnvironmentError',
       'Exception',
       'False',
       'FileExistsError',
       'FileNotFoundError',
       'FloatingPointError',
       'FutureWarning',
       'GeneratorExit',
       'IOError',
       'ImportError',
       'ImportWarning',
       'IndentationError',
       'IndexError',
       'InterruptedError',
       'IsADirectoryError',
       'KeyError',
       'KeyboardInterrupt',
       'LookupError',
       'MemoryError',
       'ModuleNotFoundError',
       'NameError',
       'None',
       'NotADirectoryError',
       'NotImplemented',
       'NotImplementedError',
       'OSError',
       'OverflowError',
       'PendingDeprecationWarning',
       'PermissionError',
       'ProcessLookupError',
       'RecursionError',
       'ReferenceError',
       'ResourceWarning',
       'RuntimeError',
       'RuntimeWarning',
       'StopAsyncIteration',
       'StopIteration',
       'SyntaxError',
       'SyntaxWarning',
       'SystemError',
       'SystemExit',
       'TabError',
       'TimeoutError',
       'True',
       'TypeError',
       'UnboundLocalError',
       'UnicodeDecodeError',
       'UnicodeEncodeError',
       'UnicodeError',
       'UnicodeTranslateError',
       'UnicodeWarning',
       'UserWarning',
       'ValueError',
       'Warning',
       'ZeroDivisionError',
       '__IPYTHON__',
       '__build_class__',
       '__debug__',
       '__doc__',
       '__import__',
       '__loader__',
       '__name__',
       '__package__',
       '__spec__',
       'abs',
       'all',
       'any',
       'ascii',
       'bin',
       'bool',
       'breakpoint',
       'bytearray',
       'bytes',
       'callable',
       'chr',
       'classmethod',
       'compile',
       'complex',
       'copyright',
       'credits',
       'delattr',
       'dict',
       'dir',
       'display',
       'divmod',
       'enumerate',
       'eval',
       'exec',
       'execfile',
       'filter',
       'float',
       'format',
       'frozenset',
       'get_ipython',
       'getattr',
       'globals',
       'hasattr',
       'hash',
       'help',
       'hex',
       'id',
       'input',
       'int',
       'isinstance',
       'issubclass',
       'iter',
       'len',
       'license',
       'list',
       'locals',
       'map',
       'max',
       'memoryview',
       'min',
       'next',
       'object',
       'oct',
       'open',
       'ord',
       'pow',
       'print',
       'property',
       'range',
       'repr',
       'reversed',
       'round',
       'runfile',
       'set',
       'setattr',
       'slice',
       'sorted',
       'staticmethod',
       'str',
       'sum',
       'super',
       'tuple',
       'type',
       'vars',
       'zip']
```

we have seen some very useful builtins like:

- range
- abs
- enumerate
- sorted
- sum

two additional builtins that are useful with the following data structures are **map** and **filter**

5 Data Structures

In the first class we ad a glimpse of *lists*. In this section we'll learn how to deal with *lists, strings, tuples, dictionaries* and *sets*.

5.1 Lists

a.k.a. the workhorse data structure of Python The most general type of data collection objects (iterables) in Python are **lists**. Lists are *ordered, mutable sequences*.

[22]:
```
empty_list = []
another_empty_list = list()
```

```
letters = ["Alpha","Bravo","Charlie","Delta","Echo","Foxtrot"]
```

You can access members of the list using the **index** of that item:

```
[23]: letters[2]
```

```
[23]: 'Charlie'
```

Counting starts from 0 (as in C and derived languages). Thus, letters is a 6 elements list whose first and last elements are:

```
[24]: print(letters[0],letters[5])
```

```
Alpha Foxtrot
```

```
[25]: letters[6]
```

```
              ␣
 ↪---------------------------------------------------------------------------

        IndexError                                Traceback (most recent call␣
 ↪last)

        /tmp/ipykernel_233018/3500960445.py in <module>
    ----> 1 letters[6]


        IndexError: list index out of range
```

Counting may start from the end of the list and go backwards:

```
[26]: print(letters,"\n",letters[-1],letters[-2])
```

```
['Alpha', 'Bravo', 'Charlie', 'Delta', 'Echo', 'Foxtrot']
 Foxtrot Echo
```

Lists are **mutable**:

```
[27]: letters[3]="Kilo"
      letters
```

```
[27]: ['Alpha', 'Bravo', 'Charlie', 'Kilo', 'Echo', 'Foxtrot']
```

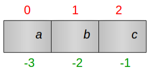
```
[28]: letters[3]="Delta"
      letters
```

```
[28]: ['Alpha', 'Bravo', 'Charlie', 'Delta', 'Echo', 'Foxtrot']
```

The layout of a list values and indexes is:

```
[29]: Image(filename="list.png")
```

[29]:



You can add additional items to the list using the **.append()** and **.insert()** methods:

```
[30]: letters.append("Hotel")
      letters
```

```
[30]: ['Alpha', 'Bravo', 'Charlie', 'Delta', 'Echo', 'Foxtrot', 'Hotel']
```

```
[31]: letters.insert(6,"Golf")
      letters
```

```
[31]: ['Alpha', 'Bravo', 'Charlie', 'Delta', 'Echo', 'Foxtrot', 'Golf', 'Hotel']
```

Different types may be mixed in lists;
lists support addition:

```
[32]: a=[0,1.1,2];b=["q","e","r"];print(a+b)
```

```
[0, 1.1, 2, 'q', 'e', 'r']
```

The **in** keyword is used to check if an item is in a list (and other iterables):

```
[33]: "Alpha" in letters
```

```
[33]: True
```

```
[34]: "Juliet" in letters
```

```
[34]: False
```

The **range** command is a convenient way to make sequential lists of numbers, **from *m* to *n-1***:

```
[35]: for i in range(2,8):
          print(i)
```

```
2
3
4
5
```

```
6
7
```

The lists created above with range have a *step* of 1 between elements. You can also give a fixed step size via a third argument:

```
[36]: evens = range(0,10,2)
      evens
```

```
[36]: range(0, 10, 2)
```

the result of **range** by itself *is not a list* but it can be used to *generate* a list (more on this later on):

```
[37]: E=list(evens);E
```

```
[37]: [0, 2, 4, 6, 8]
```

```
[38]: evens[3]
```

```
[38]: 6
```

```
[39]: E[3]
```

```
[39]: 6
```

You can find out how long a list is using the **len()** command (remember the help() function and IPython introspection:

```
[40]: len(evens)
```

```
[40]: 5
```

pop returns the last element of the list and removes it.

```
[41]: letters.pop()
```

```
[41]: 'Hotel'
```

```
[42]: letters
```

```
[42]: ['Alpha', 'Bravo', 'Charlie', 'Delta', 'Echo', 'Foxtrot', 'Golf']
```

Lists and strings have something in common: they can both be treated as sequences. You can iterate on the letters of a string as you would do with a list:

```
[43]: for letter in "Bravo":
          print(letter)
```

```
B
r
a
v
o
```

```
[44]: zulu=list("zulu")
      zulu
```

```
[44]: ['z', 'u', 'l', 'u']
```

```
[45]: "".join(zulu)
```

```
[45]: 'zulu'
```

5.1.1 Slicing

Strings and lists support the **slicing** operation, which you can also use on any sequence. We already know that we can use *indexing* to get any single element of a list:

```
[46]: letters[2]
```

```
[46]: 'Charlie'
```

If we want the list containing the first two elements of a list, we can do this via

```
[47]: letters[2:5]
```

```
[47]: ['Charlie', 'Delta', 'Echo']
```

or simply

```
[48]: letters[:2]
```

```
[48]: ['Alpha', 'Bravo']
```

If we want the last items of the list, we can do this with negative slicing:

```
[49]: letters[-2:]
```

```
[49]: ['Foxtrot', 'Golf']
```

which is somewhat logically consistent with negative indices accessing the last elements of the list.

Slicing is also supported by strings:

```
[50]: mystring="mystring";mystring[2:6:2]
```

[50]:
```
'sr'
```

Here we used a *step* in selecting the interval. The general syntax is:

[start:stop:step]

with defaults:

start = 0
stop = -1
step = 1

i.e.

mylist[:len(mylist)]

means

mylist[0:len(mylist):1]

[51]:
```
numbers = range(0,11)
evens = numbers[2::2]
list(evens)
```

[51]:
```
[2, 4, 6, 8, 10]
```

We can slice backwards from the end of the list:

[52]:
```
list(numbers[-1::-2])
```

[52]:
```
[10, 8, 6, 4, 2, 0]
```

The mechanism of slicing and indexing presented above for lists works (with little variation) with other iterables types in Python and in Numpy

List may also be *sorted*; find out more with help(sorted) or with **introspection**:

[53]:
```
sorted(list(numbers[-1::-2]))
```

[53]:
```
[0, 2, 4, 6, 8, 10]
```

[54]:
```
help(sorted)
```

```
Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customize the sort order, and the
    reverse flag can be set to request the result in descending order.
```

the custom key function may be particularly handy…

Lists within lists Lists can be regarded as eterogeneous data containers. Hence, you can create access and slice *lists of lists*:

[55]:
```
mylist = [["Alpha","Bravo","Charlie"],["X-Ray","Yankee","Zulu"]]
```

[56]:
```
mylist[-1]
```

[56]:
```
['X-Ray', 'Yankee', 'Zulu']
```

[57]:
```
mylist[-1][-1]
```

[57]:
```
'Zulu'
```

[58]:
```
mylist[-1][-1][-1]
```

[58]:
```
'u'
```

5.1.2 List comprehensions

Apply an expression to every element of a list. Can simultaneously map
(i.e. associate elements) and filter.

[59]:
```
pow2 = [2.0**i for i in range(0,8)]
#notice the range function (xrange in Python2.x) to generate the sequence
pow2
```

[59]:
```
[1.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0, 128.0]
```

[60]:
```
bases=["A","C","G","T"]
purines=list()
[ purines.append(base) for base in bases if base in ["A","G"]]
 # list comprehension with filter; notice the append method for lists
purines
```

[60]:
```
['A', 'G']
```

5.1.3 Exercises

1. Access to the element with value `Sierra' from the list [[1,2,3],[``1st'',``2nd'',``3rd''],[``Alpha'',``Sierra'',``Tango'']],"1.0]
2. Write a script which accepts a string from console and print the characters that have even indexes. If the following string is given as input to the program:

   H1e2l3l4o5w6o7r8l9d

Then, the output of the program should be:

```
Helloworld
```

3. Write a function to determine if a given string is a palyndrome.

4. Given a list of strings, return a list with the strings in sorted order, except group all the strings that begin with `x' first: [`Alpha', `Delta', `xray', `charlie'] yields [`x-ray', `alpha', `charlie', `delta'].

5.1.4 Hints

2 Use list[::2] to iterate a list with step 2. Do you have enough elements in the list?

3 Slicing.

4 sorted()

[61]:

```
#Solution #1
mylist=[[1,2,3],[["1st","2nd","3rd"],["Alpha","Sierra","Tango"]],1.0]
mylist[1][1][1]
```

[61]:

```
'Sierra'
```

[62]:

```
# Solution 2
s = input()
s = s[::2]
print(s)
```

```
Uniform
Uiom
```

[63]:

```
#Solution 3
def find_palyndrome(mystring):
    """
    a docstring here
    """
    if len(mystring)==1:
        print("are you kidding me?")
        return None
    S = mystring.lower()
    S = S.replace(" ","")
    L = len(S)
    half1 = S[:L//2]
    if L%2==0:
        half2 = S[-1:L//2-1:-1]
    else:
        half2 = S[-1:L//2:-1]
    if half1 == half2:
        return True
    else:
        return False
find_palyndrome("i topi non avevano nipoti")
```

[63]:
```
True
```

[64]:
```
#solution 4
def front_x(words):
    x_list = list()
    other_list = []
    for w in words:
        if w.startswith('x'):
            x_list.append(w)
        else:
            other_list.append(w)
    return sorted(x_list) + sorted(other_list)
front_x(['Alpha', 'Delta', 'xray', 'charlie'])
```

[64]:
```
['xray', 'Alpha', 'Delta', 'charlie']
```

5.2 Strings

[65]:
```
#create astring
mystring  = "Alpha Bravo"
mystring[10]
```

[65]:
```
'o'
```

[66]:
```
#from a string to list
mylist=list(mystring)
mylist
```

[66]:
```
['A', 'l', 'p', 'h', 'a', ' ', 'B', 'r', 'a', 'v', 'o']
```

what if we want a list of words? we can use string associated methods

[67]:
```
mylist = mystring.split()
mylist
```

[67]:
```
['Alpha', 'Bravo']
```

split separates a string into substrings using as a separator a character, by default a whitespace

alternative ways of creating strings:

[68]:
```
double_quotes="Alpha Bravo Charlie"
```

```
[69]: single = 'Alpha Bravo Charlie'
```

```
[70]: double_quotes is single
```

```
[70]: False
```

```
[71]: double_quotes == single
```

```
[71]: True
```

```
[72]: triple_single ='''Alpha Bravo Charlie'''
      triple_double = """Alpha Bravo Charlie"""
```

pay attention to the type of quotes being used:

```
[73]: "A single quote (') inside a double quote"
```

```
[73]: "A single quote (') inside a double quote"
```

```
[74]: 'Here we have "double quotes" inside single quotes'
```

```
[74]: 'Here we have "double quotes" inside single quotes'
```

```
[75]: "You cannot mix quotes'
```

```
          File "/tmp/ipykernel_233018/2716310145.py", line 1
        "You cannot mix quotes'
                               ^
    SyntaxError: EOL while scanning string literal
```

Strings are **immutable**, at variance with lists:

```
[76]: s="alpha"
      s
```

```
[76]: 'alpha'
```

```
[77]: print(s[0])
      s[0] = "A"
```

```
a

      ␣
↪---------------------------------------------------------------------------
```

```
        TypeError                                 Traceback (most recent call␣
 ↪last)

        /tmp/ipykernel_233018/1735615679.py in <module>
          1 print(s[0])
    ----> 2 s[0] = "A"


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

5.3 Tuples

A **tuple** is a sequence object like a list or a string. It's constructed by grouping a sequence of objects together with commas, either without brackets, or with parentheses. At variance with *list*, tuples are **immutable**, they don't have *append, insert* or *pop* methods.

```
[78]: t = ("tinker","taylor","soldier","spy")
      t
```

```
[78]: ('tinker', 'taylor', 'soldier', 'spy')
```

```
[79]: t[0]
```

```
[79]: 'tinker'
```

```
[80]: t[-1]
```

```
[80]: 'spy'
```

```
[81]: T = tuple()
      T = T + t
      T
```

```
[81]: ('tinker', 'taylor', 'soldier', 'spy')
```

```
[82]: T[3]="sailor"
```

```
      ␣
 ↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
 ↪last)

        /tmp/ipykernel_233018/1182022236.py in <module>
    ----> 1 T[3]="sailor"
```

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

5.3.1 Exercise

Given a list of non-empty tuples, return a list sorted in increasing order by the last element in each tuple.

[(1, 7), (1, 3), (3, 4, 5), (2, 2)]

yields

[(2, 2), (1, 3), (3, 4, 5), (1, 7)]

5.3.2 Hint

Hint: use a custom key= function to extract the last element form each tuple.

[83]:

```
def yield_last(a):
    return a[-1]

def sort_last(tuples):
    return sorted(tuples, key=yield_last)
sort_last( [(1, 7), (1, 3), (3, 4, 5), (2, 2)])
```

[83]:

```
[(2, 2), (1, 3), (3, 4, 5), (1, 7)]
```

5.4 Dictionaries

A unordered mapping of keys to values. Associate a key with a value. Each key must be unique. Keys and values may be of any type and may be mixed.

The **keys()** and **values()** methods return *list generators*; list(d.keys()) will give you a list. The **items()** method returns a *a generator of a list of tuples* of length 2: (key:value). Why tuples?

[ ]:

```
Image(filename="dict.png")
```

[ ]:

```
a=dict();b= {};c={0:"q",1:"w"}
print(a,b,c)
```

[ ]:

```
a['K'] = 'lysine'
a['P']='proline'
a
```

[ ]:

```
print("KEYS: ",a.keys(),"VALUES: ",a.values())
```

[ ]:

```
a.keys()[0]
```

[ ]:

```
list(a.keys())
```

[ ]:

```
a.items()
```

You can check the presence of a *key*:

[ ]:

```
'K' in a
```

[ ]:

```
'Q' in a
```

[ ]:

```
'lysine' in a
```

being mutable, you can eliminate items from dicts, using **del**

[ ]:

```
del a['K']
a
```

Dictionaries can be used in a function definition to define *optional keyword arguments* creating very flexible interfaces:

[ ]:

```
import math
def distance(**kwargs):
    for key, value in kwargs.items():
        if key is "def":
            metric = value
        if key is "v1":
            v1 = value
        if key is "v2":
            v2 = value
    if len(v1) != len(v2):
        return False
    dist = .0
    if metric is "euclidean":
        for i in len(v1):
            dist = dist + (v1[i]-v2[i])**2
        dist = math.sqrt(dist)
    elif metric is "cityblock":
        for i in len(v1):
            dist = dist + abs(v1[i])+abs(v2[i])
    else:
        print("I miss this definition")
        return None
```

5.4.1 Exercises

1. Write a program that accepts a sentence and calculate the number of letters and digits:

   hello, world! 123

```
LETTERS 10 DIGITS 3
```

2. Given a dictionary **d** and a key **k**, it is easy to find the corresponding value **v = d[k]**. This operation is called a lookup. But what if you have v and you want to find **k**? You have two problems: first, there might be more than one key that maps to the value v. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a reverse lookup; you have to search. Write a function that performs a reverse lookup and test it.

3. Look at the txt file hivsequences.txt; it contains several HIV sequences. Write a program to load each of them in a appropriate container and then for each file calculate the proportion of each base in each sequence. Before using the actual file it may be wise test your program with something like that (You may want to write a script and use %run):

```
Seq1 AAAACCCGGT
Seq2 AGCTACGTATA
Seq3 AGCTACGTATA
```

[84]:
```
s=!head -1 hivsequences.txt
s[0][:300]
```

[84]:
```
'HIV-1.A ATGGGTGCGAGAGCGTCAATATTAAGCGGGGGAAGATTAGATGCATGGGAGAAAATTCGGCTAAGGCCAGG
GGGAAAGAAAAAATATAGACTAAAACATCTAGTATGGGCAAGCAGGGAGCTGGAGAGATTCGCACTTAAYCCTRGCCTTT
TAGAATCAGCAGAAGGATGTCAACAAATAATGGAACAGTTACAACCAGCTCTYAAGACAGGAWCAGAAGAAATTAAATCA
TTATTTAATACAGTAGCAACCCTCTATTGTGTACATCAAAGGATAGATGTAAAAGACACCA'
```

[85]:
```
# Solution 1
s = input()
d={"DIGITS":0, "LETTERS":0}
for c in s:
    if c.isdigit():
        d["DIGITS"]+=1
    elif c.isalpha():
        d["LETTERS"]+=1
print("LETTERS", d["LETTERS"])
print("DIGITS", d["DIGITS"])
```

```
H3ll0 W0rld1
LETTERS 7
DIGITS 4
```

[86]:
```
#Solution 2
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
# a simple print may make your day
```

```
    raise ValueError

#what if I have more than one key mapping the same value?
```

[87]:
```
#Solution 3
%run sequence.py hivsequences.txt
```

```
there are  16  sequences in the file
Composition of sequence  HIV-1.A_0  is
A 0.3710626514364832
C 0.17364716741663783
G 0.2342217607015115
T 0.2188761970693435
Composition of sequence  HIV-1.A_1  is
A 0.3512392409001348
C 0.18189360157627293
G 0.24452971067095303
T 0.22233744685263923
Composition of sequence  HIV-1.A_2  is
A 0.35500515995872034
C 0.17760577915376677
G 0.2436532507739938
T 0.2237358101135191
Composition of sequence  HIV-1.A_3  is
A 0.3612582781456954
C 0.178476821192053
G 0.2390728476821192
T 0.22119205298013245
Composition of sequence  HIV-1.B_4  is
A 0.35096203313098057
C 0.18232328428850705
G 0.24416092190554584
T 0.22255376067496657
Composition of sequence  HIV-1.B_5  is
A 0.35919632360799403
C 0.17580421075130917
G 0.23917922411029177
T 0.22411029176017955
Composition of sequence  HIV-1.B_6  is
A 0.36227212094264116
C 0.17474433081369498
G 0.24132947976878613
T 0.22154290795909293
Composition of sequence  HIV-1.B_7  is
A 0.36427439438532944
C 0.1785148290695042
G 0.23647271904007244
```

```
T 0.22039846049354767
Composition of sequence  HIV-1.C_8  is
A 0.36566581091639694
C 0.17356847862484653
G 0.24054023886594486
T 0.2202254715928117
Composition of sequence  HIV-1.C_9  is
A 0.3620861477134315
C 0.1789392093898793
G 0.2370723064998339
T 0.22190233639685528
Composition of sequence  HIV-1.C_10  is
A 0.36469673405909797
C 0.17385025549877806
G 0.23916907353921352
T 0.22228393690291046
Composition of sequence  HIV-1.C_11  is
A 0.36133614471201864
C 0.17778271002108534
G 0.24126068138941295
T 0.21962046387748307
Composition of sequence  HIV-1.D_12  is
A 0.3632301656495205
C 0.17785527462946818
G 0.2374673060156931
T 0.22144725370531823
Composition of sequence  HIV-1.D_13  is
A 0.3694951664876477
C 0.17376775271512113
G 0.23284401479890202
T 0.22389306599832914
Composition of sequence  HIV-1.D_14  is
A 0.36752136752136755
C 0.17413105413105412
G 0.23487179487179488
T 0.22347578347578348
Composition of sequence  HIV-1.D_15  is
A 0.3668453976764969
C 0.17605004468275245
G 0.23782394995531725
T 0.21928060768543342
```

5.5 Sets

A **set** is an unordered, mutable sequence of data with no duplicate elements. It has the form:

[88]:

```
not_a_set=[0,1+1j,"foo","foo",1+1j]
print(not_a_set)
myset = set(not_a_set)
myset
```

```
[0, (1+1j), 'foo', 'foo', (1+1j)]
```

[88]:

```
{(1+1j), 0, 'foo'}
```

remember that they are *unordered*:

[89]:

```
myset[0]
```

```
       ␣
↪---------------------------------------------------------------------------

     TypeError                                 Traceback (most recent call␣
↪last)

     /tmp/ipykernel_233018/435093538.py in <module>
   ----> 1 myset[0]


     TypeError: 'set' object is not subscriptable
```

They are mostly used for membership testing and to eliminate duplicate entries. Sets support mathematical operations like:

1. **union**: The returned set contains the elements of all sets.
2. **intersection**: The returned set contains the elements common to all sets.
3. **difference**: The returned set contains the elements of the first set, which are not in the others.
4. **symmetric difference**: The returned set contains elements, which are present in only one set.

[90]:

```
another_set={0,1+1j,"bar"}
```

[91]:

```
another_set.union(myset)
```

[91]:

```
{(1+1j), 0, 'bar', 'foo'}
```

Argument may be a list, tuple or string:

[92]:

```
mytuple=("foo","bar")
mytuple
```

```
[92]: ('foo', 'bar')
```

[93]:
```
another_set.difference(mytuple)
```

```
[93]: {(1+1j), 0}
```

[94]:
```
another_set.symmetric_difference(myset)
```

```
[94]: {'bar', 'foo'}
```

5.5.1 Exercise

Given a list of words (buzfoo, foobar, barbuz, buzbuz) find the minimum set of lenght=3 prefixes and suffixes (foo, bar, buz)

[95]:
```
#Solution
def find_min(words):
    prefix = list()
    suffix = list()
    for w in words:
        prefix.append(w[:3])
        suffix.append(w[-3:])
    prefix = set(prefix)
    return prefix.union(suffix)
words = ("foofoo","bazbar","barbaz","foobar","xyzxyz","qwertyu")
find_min(words)
```

```
[95]: {'bar', 'baz', 'foo', 'qwe', 'tyu', 'xyz'}
```

5.6 Data structures summary

-- Lists, sets and dictionaries are extensible and mutable.

-- Tuples, on the other hand:

-- Not extensible

-- Values cannot be modified

-- Dictionaries and sets are unordered.

-- Elements of sets are not indexed.

Tuples require less storage and are treated faster by the interpreter.

Since sets have unique elements, the interpreter can optimize membership tests.

[96]:
```
language  = "Python" #a string
horses = ["king","soldatino","dartagnan"] # a list
numbers = (3,1,1,2,2,3) # a tuple
dna = {"A":"adenine","C":"cytosine","G":"guanosine","T":"thymine"} #a dictionary
```

```
unique_numbers  = set(numbers)
print(unique_numbers)
```

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

5.7 map and filter

From the documentation:

filter(function=None, iterable): Construct an iterator from those elements of iterable for which function returns True. The iterable may be either a sequence, a container which supports iteration, or an iterator. If function is None, the identity function is assumed, that is, all elements of iterable that are False are removed.

map(function,iterable): Construct an iterator that applies function to every item of iterable, yielding the results. If additional iterable arguments are passed, function must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted.

An iterable is any python python object that supports iteration, such as strings, lists or user defined data structures. An iterator is an object that runs over the ``components'' with of iterable one at time until a **StopIteration** is raised.

This will be clearer when OOP will be presented.

[97]:
```
mylist = list("0123456789")
mylist
```

[97]:
```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

[98]:
```
mylist = map(int,mylist)
mylist, type(mylist)
```

[98]:
```
(<map at 0x7f12b048ed90>, map)
```

[99]:
```
mylist.__next__()
```

[99]:
```
0
```

[100]:
```
list(mylist)
```

[100]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

What happened? map returned a **iterator object** that applies int to every element in mylist, when *invoked* with **__next__** or list(). Python2 map used to return directly a list:

[101]:
```
%%python2
a=map(int,list("012345678"))
print a, type(a)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8] <type 'list'>
```

[102]:
```
import random
def head_tail(num):
    num = int(num)
    if random.random() > 0.5:
        return 1
    else:
        return 0
mylist = list("0123456789")
mylist = list(map(head_tail,mylist))
mylist
```

[102]:
```
[1, 0, 1, 1, 0, 0, 1, 1, 1, 1]
```

[103]:
```
tuple(filter(None,mylist))
```

[103]:
```
(1, 1, 1, 1, 1, 1, 1)
```

[104]:
```
def play_only_red(a):
    reds =␣
 ↪list(range(1,11,2))+list(range(19,29,2))+list(range(12,19,2))+list(range(20,37,2))
    return a in reds
```

[105]:
```
games = [random.randint(1,36) for i in range(20)] # a list comprehension
print(games)
tuple(filter(play_only_red,games))
```

```
[9, 29, 17, 13, 18, 11, 20, 16, 29, 25, 19, 15, 23, 33, 30, 20, 7, 32, 11, 4]
```

[105]:
```
(9, 18, 20, 16, 25, 19, 23, 30, 20, 7, 32)
```

The **lambda** operator allows us to build *anonymous functions*, which are simply functions that aren't defined by a normal **def** statement with a name. For example, a function that triples the input is:

[106]:
```
def triplex(x): return 3*x
triplex(3)
```

[106]:
```
9
```

[107]:
```
triplex = lambda x: 3*x
triplex(3)
```

[107]: 9

what's the point of this? **lambda** is particularly useful when you need a simple throw away function to be used in combination with **map** or **filter**:

[108]:
```
tuple(filter(lambda x: x%2==0,games))
```

[108]: (18, 20, 16, 30, 20, 32, 4)

5.8 Exercise

Write a function filter_long_words() that takes a list of words and an integer n and returns the list of words that are longer than n

[109]:
```
#Solution:
words = ["short","loooooong","veeeeeryyyy looooonnnnngggg"]
def flong(tofilt,maxl = 6):
    return tuple(filter(lambda x: len(x)>maxl,tofilt))
print(flong(words))
```

```
('loooooong', 'veeeeeryyyy looooonnnnngggg')
```

6 Regular expressions

Regular expressions are a powerful string manipulation tool. All modern languages have similar library packages for regular expressions Use regular expressions to: 1. Search a string (search and match) 2. Replace parts of a string (sub) 3. Break strings into smaller pieces (split)

The two basic functions are **re.search** and **re.match**.

Search looks for a pattern anywhere in a string.

Match looks for a match staring at the beginning.

Both return None (logical false) if the pattern isn't found and a ``match object'' instance if it is. Different matches may be referred to using groups.

[110]:
```
import re
pat = "a*b"
SE = re.search(pat,"fooaaabcde")
SE
```

[110]: <re.Match object; span=(3, 7), match='aaab'>

[111]:
```
MA = re.match(pat,"fooaaabcde")
print(MA)
```

```
None
```

```
[112]: SE.group(0)
```

```
[112]: 'aaab'
```

```
[113]: pat2 = '(\w+)\@((\w+\.)+(com|org|net|edu))' # notice the wildcards!!
        r2 = re.match(pat2,"finin@cs.umbc.edu")
        r2.group(1)
```

```
[113]: 'finin'
```

```
[114]: r2.groups()
```

```
[114]: ('finin', 'cs.umbc.edu', 'umbc.', 'edu')
```

Common regular expression syntax:

- . Matches any char but newline (by default)
- ^ Matches the start of a string
- $ Matches the end of a string
- * Any number of what comes before this
- \+ One or more of what comes before this
- | Or
- \w Any alphanumeric character
- \d Any digit
- \s Any whitespace character
- \W matches NON-alphanumeric, \D NON digits, etc
- [aeiou] matches any of a, e, i, o, u

6.1 Exercises

1. Why in ``finin@cs.umbc.edu'' we have groups `cs.umbc.edu' and `umbc.' in the regular expression above?
2. A website requires the users to input username and password to register. Write a program to check the validity of password input by users. Following are the criteria for checking the password:
    1. At least 1 letter between [a-z]
    2. At least 1 number between [0-9]

3. At least 1 letter between [A-Z]
4. At least 1 character from [!$#@]
5. Minimum length of transaction password: 6
6. Maximum length of transaction password: 12

Your program should accept a sequence of comma separated passwords and will check them according to the above criteria. Passwords that match the criteria are to be printed, each separated by a comma. If the following passwords are given as input to the program:

ABd1234@1,a AF1#,2w3E* ,2We3345

Then, the output of the program should be:

ABd1234@1

[115]:

```
#Solution
import re
value=list()
pwlist = input("Enter password list: ").split(",")
for p in pwlist:
    if len(p)<6 or len(p)>12:
        continue
    else:
        pass
    if not re.search("[a-z]",p):
        continue
    elif not re.search("[0-9]",p):
        continue
    elif not re.search("[A-Z]",p):
        continue
    elif not re.search("[!$#@]",p):
        continue
    elif re.search("\s",p):
        continue
    else:
        pass
    value.append(p)
value = ",".join(value)
print(value)
```

```
Enter password list: ABd1234@1,a AF1#,2w3E* ,2We3345
ABd1234@1
```

7 Variable scope

When a variable a is created:

a = ["q","w"]

you associate (**bind**) ``a'' name to a memory location holding the tuple (q,w). You can later use a in another contaniner, such as a dictionary:

```
[116]: alist = ["q","w"]
       another_list = alist
       a_list_of_lists =[alist,another_list]
       print(alist,another_list,a_list_of_lists)
```

```
['q', 'w'] ['q', 'w'] [['q', 'w'], ['q', 'w']]
```

What happens to another_list and a_list_of_list if I change alist? - another_list stays the same or is aware of the change? - a_list_of_lists stays the same or is aware of the change?

```
[117]: alist = ["e","r"]
       alist
```

```
[117]: ['e', 'r']
```

```
[118]: another_list
```

```
[118]: ['q', 'w']
```

binding alist to different value does not affect another_list
what about `a_list_of_lists'?

```
[119]: a_list_of_lists
```

```
[119]: [['q', 'w'], ['q', 'w']]
```

but, if I change something **in** a list …

```
[120]: alist = ["q","w"]
       another_list = alist
       a_list_of_lists =[alist,another_list]
       alist.pop()
```

```
[120]: 'w'
```

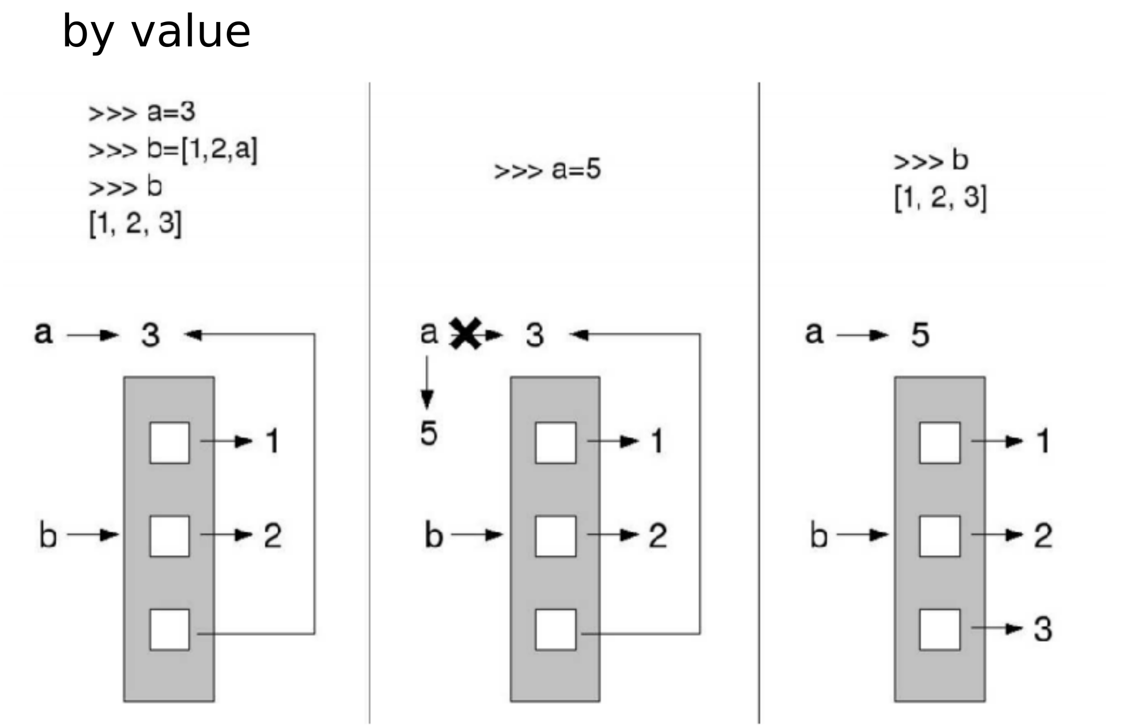
```
[121]: print(alist,another_list,a_list_of_lists)
```

```
['q'] ['q'] [['q'], ['q']]
```

what happened?

```
[122]: Image(filename="byvalue.png")
```

```
[122]:
```



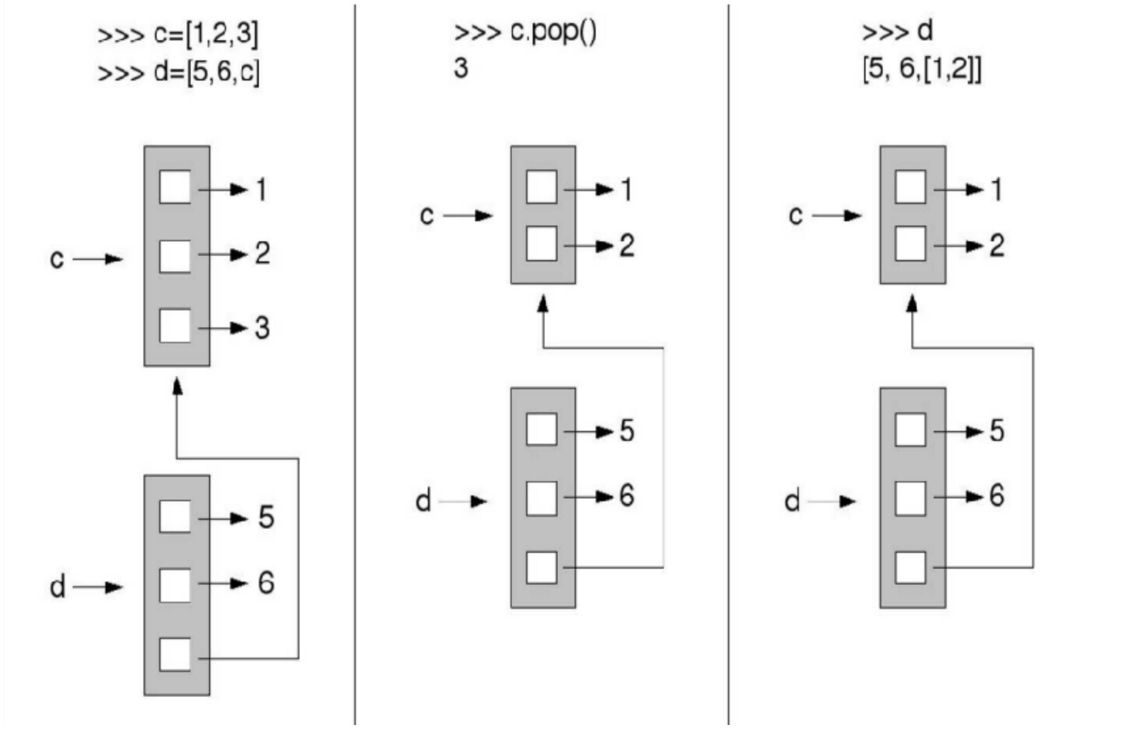
```
>>> a=3
>>> b=[1,2,a]
>>> b
[1, 2, 3]
```

```
>>> a=5
```

```
>>> b
[1, 2, 3]
```

```
[123]: Image(filename="byref.png")
```

[123]:



```
>>> c=[1,2,3]
>>> d=[5,6,c]
```

```
>>> c.pop()
3
```

```
>>> d
[5, 6,[1,2]]
```

How we can copy alist into another_list?

[124]:
```
alist = ["q","w"]
another_list = alist[:]
alist.pop()
alist,another_list
```

[124]:
```
(['q'], ['q', 'w'])
```

define some variables:

[125]:
```
myvar = 3
mystring = "charlie"
mylist = list(mystring)
print(myvar,mystring,mylist)
```

```
3 charlie ['c', 'h', 'a', 'r', 'l', 'i', 'e']
```

[126]:
```
def manipulate_some_variables(avar,astring,alist):
    bvar = avar
    bvar += 1
    bstring = astring
    bstring = bstring.upper()
    blist = alist
    blist.pop()
    return bvar,bstring,blist
myvar2,mystring2,mylist2 = manipulate_some_variables(myvar,mystring,mylist)
```

[127]:
```
print(myvar,mystring,mylist,"\n",myvar2,mystring2,mylist2)
```

```
3 charlie ['c', 'h', 'a', 'r', 'l', 'i']
 4 CHARLIE ['c', 'h', 'a', 'r', 'l', 'i']
```

The function works on **local** copies of variables and **myvar** and **mystring**; the third argument however is affected; why?

In Fortran whenever a subroutine or function is called you can modified it locally because the information passed between the **caller** and the **callee** is a **pointer** to the variable. In C you pass *copies of variables* but you can modify them locally by passing **pointers**.

In Python the actual behaviour depends on the nature of the variable being passed. For **immutable** objects what is passed is actually a **copy**, i.e. the caller is unaware of any local modification done by the callee. **Mutable** objects however are affected by changes in the callee.

In Fortran, you may write something like:

```
  program callts
  Implicit real*8(a-h,o-z)
  V1 = 1.d0
  V2 = 2.d0
  Call reverse(V1,V2)
  write(*,*) V1, V2
  end program callts

  Subroutine reverse(var1,var2)
  Implicit real*8(a-h,o-z)
  tmp  = var1
  var1 = var2
  var2 = tmp
  Return
  End
```

```
gmancini@shangrila:~\$ gfortran  callts.f
gmancini@shangrila:~\$ ./a.out
2.0000000000000000        1.0000000000000000
```

in C:

```
#include "stdio.h"

void reverse_pnt(float* var1, float* var2){
   float var3 = *var1;
   *var1 = *var2;
   *var2 = var3;
   printf("ByRef, Callee %p %p\n",var1,var2);
   }

void reverse_val(float var1, float var2){
   float var3 = var1;
   var1 = var2;
   var2 = var3;
   printf("ByVal, Callee %p %p\n",&var1,&var2);
   }

void main(int argc, char** argv){
    float V1, V2;
    V1 = 1.0;
    V2 = 2.0;
    printf("Caller address %p %p\n",&V1,&V2);
    printf("Caller values %f %f\n",V1,V2);
    reverse_val(V1,V2);
    printf("Result %f %f\n",V1,V2);
    reverse_pnt(&V1,&V2);
    printf("Result %f %f\n",V1,V2);
    }
```

```
gmancini@tortillaflat:~$ gcc pass.c
gmancini@tortillaflat:~$ ./a.out
Caller address 0x7ffd153aa240 0x7ffd153aa244
Caller values 1.000000 2.000000
ByVal, Callee 0x7ffd153aa20c 0x7ffd153aa208
Result 1.000000 2.000000
ByRef, Callee 0x7ffd153aa240 0x7ffd153aa244
Result 2.000000 1.000000
```

[128]:

```python
def mutate(a,b,c,d):
    a=2
    b.append(0)
    c = c+"s"
    d = d+[0,1]
    return a,b,c,d
```

[129]:

```python
a = 0
b = ["q","w"]
c = "string"
d = [2,3]
mutate(a,b,c,d)
print(a,b,c,d)
```

```
0 ['q', 'w', 0] string [2, 3]
```

[130]:

```python
help(id)
```

```
Help on built-in function id in module builtins:

id(obj, /)
    Return the identity of an object.

    This is guaranteed to be unique among simultaneously existing objects.
    (CPython uses the object's memory address.)
```

[131]:

```python
def mutate(a,b,c,d):
    a=2
    b.append(0)
    c = c+"s"
    d = d+[0,1]
    print(id(a),id(b),id(c),id(d))
    return a,b,c,d
```

[132]:

```python
a = 0
b = ["q","w"]
c = "string"
```

```
d = [2,3]
mutate(a,b,c,d)
print(id(a),id(b),id(c),id(d))
```

```
9789024 139718244188416 139718243395760 139718244105152
9788960 139718244188416 139718366762416 139718295119872
```

IMMUTABLE objects are ``passed by **value**'' while **MUTABLE** objects are ``passed by **reference**''. Neither definition is strictly true; both types of variable are more properly **names** associated to **objects** created at the moment of their creation. Within the function references to these **objects** are passed. But **immutable** objects names are associated to new **objects** that are valid only within the function. A function that can modify its arguments is called a *modifier* otherwise is a *pure function*. Some programming languages (e.g. *Scheme*) forbid mutable data and changes in place and are called *functional languages*.

The module **copy** allows to create copies of mutable objects:

[134]:
```
import copy
list1 = ["King","Soldatino","D\'Artagnan"]
list2 = copy.copy(list1)
list1.pop()
print(list1,list2)
```

```
['King', 'Soldatino'] ['King', 'Soldatino', "D'Artagnan"]
```

What if I have a mutable object made up of mutable objects?

[135]:
```
chemicals =␣
 ↪[["Caffeine","Theobromine","Theophylline"],["coffee","cacao","guarana"]]
xantines = copy.copy(chemicals)
chemicals[0].pop()
xantines
```

[135]:
```
[['Caffeine', 'Theobromine'], ['coffee', 'cacao', 'guarana']]
```

[136]:
```
xantines = copy.deepcopy(chemicals)
chemicals[0].append("Theophylline")
xantines
```

[136]:
```
[['Caffeine', 'Theobromine'], ['coffee', 'cacao', 'guarana']]
```

The **copy** method provides a *shallow copy*, taking references to objects referred to in the original names while **deepcopy** allows to make ``hard'' copies of everything:

[137]:
```
three_lists = [[[1]]]
tl2 = copy.deepcopy(three_lists)
three_lists[0][0].append(1)
```

```
print(three_lists,tl2)
```

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

8 Namespaces

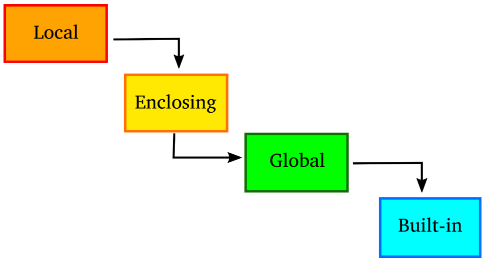
A namespace is collection of associations of names and objects belonging to a function, class or module. From the docs:

``Namespaces can be referred to as mappings associating objects and names in a given context: we have multiple independent namespaces in Python, and names can be reused for different namespaces.''

Namespace precedence:

[138]:
```
Image(filename="scope.png")
```

[138]:



1. Local can be inside a function or class method, for example.
2. Enclosing can be its enclosing function, e.g., if a function is wrapped inside another function.
3. Global refers to the uppermost level of the executing script itself, and
4. Built-in are special names that Python reserves for itself.

to access an object in a namespace:

<namespace>.<object name>

such as:

math.sqrt

What happens if I define a function inside another one?

[139]:
```
def outside(arg1,arg2):
    """
    test namespace rules in outer function
    """
    def inside(arg1,arg2):
        """
        test namespace rules in inner function
        """
        arg2 = arg2.upper()
        global arg3
        arg3 = "Charlie"
        print("local",arg1,arg2,arg3)
    inside(arg1,arg2)
    print("enclosing",arg1,arg2)
```

[140]:

```
arg1 = "Alpha"
arg2 = "Bravo"
outside(arg1,arg2)
print(arg3)
```

```
local Alpha BRAVO Charlie
enclosing Alpha Bravo
Charlie
```

the **global** statement allows to make outside aware of the creation of arg3

Do you notice anything dangerous?

global has made arg3 visible across the *module* namespace not *just inside* outer using **nonlocal** on a variable defined above allows to modify it in the enclosing scope:

[141]:

```
del arg3
def outside(arg1,arg2):
    """
    test namespace rules in outer function
    """
    arg3 = "Delta"
    def inside(arg1,arg2):
        """
        test namespace rules in inner function
        """
        nonlocal arg3
        arg2 = arg2.upper()
        arg3 = arg3.upper()
        print("local",arg1,arg2,arg3)
    inside(arg1,arg2)
    print("enclosing",arg1,arg2,arg3)
arg1 = "Alpha"
arg2 = "Bravo"
outside(arg1,arg2)
print("global",arg3)
```

```
local Alpha BRAVO DELTA
enclosing Alpha Bravo DELTA
```

```
      ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       /tmp/ipykernel_233018/2651073243.py in <module>
        18 arg2 = "Bravo"
        19 outside(arg1,arg2)
   ---> 20 print("global",arg3)


       NameError: name 'arg3' is not defined
```

9 Recursion

Functions can also call themselves, something that is called *recursion*. We're going to experiment with recursion by computing the factorial function. The factorial is defined for a positive integer n as

$$n! = n(n-1)(n-2)\cdots 1$$

First, note that we don't need to write a function at all, since this is a function built into the standard math library. Let's use some **introspection**:

[142]:
```
from math import factorial
?factorial
```

[143]:
```
factorial(10)
```

[143]: 3628800

[144]:
```
def fact(n):
    if n <= 0:
        return 1
    return n*fact(n-1)
```

[145]:
```
fact(10)
```

[145]: 3628800

9.1 Exercise

1. The formula by Srinivasa Ramanujan (see also Section ??) can be used to estimate π:

$$\frac{1}{pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k!)(1103 + 26390k)}{(k!)^4 (396)^{4k}}$$

   using a while loop and the definition of factorial above, implement it with a convergence criterion based on the value of the last term computed of 1e-15; test against math.pi.

2. Write a function able to find the greatest common divisor of positive integers I_1 and I_2. Try different implementations and compare them; start with a non iterative solution. Hint: https://en.wikipedia.org/wiki/Euclidean_algorithm

[146]:

```
# Solution 1
# credits to http://greenteapress.com/wp/think-python/
def estimate_pi():
    """Computes an estimate of pi.

    Algorithm due to Srinivasa Ramanujan, from
    http://en.wikipedia.org/wiki/Pi
    """
    total = 0
    k = 0
    factor = 2 * math.sqrt(2) / 9801
    while True:
        num = fact(4*k) * (1103 + 26390*k)
        den = fact(k)**4 * 396**(4*k)
        term = factor * num / den
        total += term

        if abs(term) < 1e-15: break
        k += 1

    return 1 / total
```

[149]:

```
estimate_pi()
```

[149]:

```
3.141592653589793
```

[151]:

```
#Solution 2
def gcd1(a, b):
    while a-b:
        a, b = b, a - b
    return a
```

[152]:

```
gcd1(80,48)
```

[152]: 16

```
[153]: #Solution 3
       def gcd2(a, b):
           if a == b:
               return a, b
           else:
               return gcd2(max(a-b,b), min(a-b,b))
```

```
[154]: gcd2(1127, 161)
```

```
[154]: (161, 161)
```

```
[155]: #Solution 4
       def gcd3(a, b):
           if b == 0:
               return a, 0
           else:
               return gcd3(b,a%b)
```

```
[156]: gcd3(1071, 462)
```

```
[156]: (21, 0)
```

10 Handling errors

10.1 Type of errors

Syntax -- wrong grammar, i.e., breaking the rules of how to write the language, e.g. forgetting punctuation, misspelling a keyword …

The program will not run at all with syntax errors

Logic - the program runs, but does not produce the expected results. Using an incorrect formula, incorrect sequence of statements, etc.

From the documentation:

Syntax errors, also known as **parsing errors**, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1, in ?
    while True print('Hello world')
               ^
```

SyntaxError: invalid syntax

The parser repeats the offending line and displays a little `arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the keyword print, since a colon (`:') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
[157]: 1/0
```

```
       ␣
↪---------------------------------------------------------------------------

      ZeroDivisionError                         Traceback (most recent call␣
↪last)

      /tmp/ipykernel_233018/2354412189.py in <module>
  ----> 1 1/0


      ZeroDivisionError: division by zero
```

```
[158]: unassigned_var
```

```
       ␣
↪---------------------------------------------------------------------------

      NameError                                 Traceback (most recent call␣
↪last)

      /tmp/ipykernel_233018/4200631271.py in <module>
  ----> 1 unassigned_var


      NameError: name 'unassigned_var' is not defined
```

Here we have three types ob **built-in** exceptions; there are more of them:

IOError
ArithmeticError
MemoryError
OSError
IndentationError
UnboundLocalError

The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not mandatory. Standard exception names are built-in identifiers (not reserved keywords):

[159]:
```
ZeroDivisionError=10
ZeroDivisionError
```

[159]: 10

[160]:
```
%reset
```

Once deleted, variables cannot be recovered. Proceed (y/[n])? y

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

Exception is the parent name of any exception type. Error handling is not limited to standard exception but may be used in any context with the **raise** keyword:

[161]:
```
a=input()
#a must be a float
if type(a) is not float:
    raise TypeError
```

10

```
            ␣
 ↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
 ↪last)

        /tmp/ipykernel_233018/2154626264.py in <module>
          2 #a must be a float
          3 if type(a) is not float:
    ----> 4     raise TypeError


        TypeError:
```

You can define custom exceptions by creating subclasses of **Exception** or simply (but less flexible) by rising built-in ones.

Once you have a defined a custom exception, it is possible to define a different behaviour from just terminating the program. A simple way to do it is by using the **try/except** construct:

[162]:
```
a=(0,1)
try:
    a[0]=2
except TypeError:
    print("object is immutable")
```

```
object is immutable
```

[163]:
```
a=(0,1)
try:
    a[0]=2
except TypeError as e:
    print(e)
```

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

[164]:
```
a=[0,1]
try:
    a[2]=2
except (TypeError,IndexError) as e:
    print("Catching multiple exceptions")
```

```
Catching multiple exceptions
```

The series of except statements may be completed with a **finally** statement which includes instructions that will be executed if an exception is raised in any block:

[165]:
```
a=[0,1]
b=("q","w")
try:
    a[2] = 2
    b[2] = "e"
except (TypeError,IndexError):
    b = list(b)
    a.append(0)
    b.append(0)
finally:
    a[2] = 2
    b[2] = "e"
    print(a,b)
```

```
[0, 1, 2] ['q', 'w', 'e']
```

10.2 Exercises

Get used to scope rules: what is the output of following snippets? Guess it before executing

```
var = 'foo'
def ex2():
    var = 'bar'
    print 'inside the function var is ', var

ex2()
print 'outside the function var is ', var


var = 'foo'
def ex3():
    global var
    var = 'bar'
    print 'inside the function var is ', var

ex3()
print 'outside the function var is ', var
```

Look at the two files ``baby names'' in the shared folder. They contain the popularity of male and female names for babies born in 2006 and 2010.

```
[166]: %%bash
       awk '/Jacob/' 02/baby2008.html
```

```
awk: fatal: cannot open file `02/baby2008.html' for reading (No such file or
directory)


       ␣
 ↪---------------------------------------------------------------------------

        CalledProcessError                        Traceback (most recent call␣
 ↪last)

        /tmp/ipykernel_233018/933818076.py in <module>
    ----> 1 get_ipython().run_cell_magic('bash', '', "awk '/Jacob/' 02/baby2008.
 ↪html\n")


        /usr/local/lib/python3.8/dist-packages/IPython/core/interactiveshell.py␣
 ↪in run_cell_magic(self, magic_name, line, cell)
       2404             with self.builtin_trap:
       2405                 args = (magic_arg_s, cell)
    -> 2406                 result = fn(*args, **kwargs)
```

```
     2407             return result
     2408


      /usr/local/lib/python3.8/dist-packages/IPython/core/magics/script.py in named_script_magic(line, cell)
      140             else:
      141                 line = script
  --> 142             return self.shebang(line, cell)
      143
      144         # write a basic docstring:


      /usr/local/lib/python3.8/dist-packages/decorator.py in fun(*args, **kw)
      230             if not kwsyntax:
      231                 args, kw = fix(args, kw, sig)
  --> 232             return caller(func, *(extras + args), **kw)
      233     fun.__name__ = func.__name__
      234     fun.__doc__ = func.__doc__


      /usr/local/lib/python3.8/dist-packages/IPython/core/magic.py in <lambda>(f, *a, **k)
      185     # but it's overkill for just that one bit of state.
      186     def magic_deco(arg):
  --> 187         call = lambda f, *a, **k: f(*a, **k)
      188
      189         if callable(arg):


      /usr/local/lib/python3.8/dist-packages/IPython/core/magics/script.py in shebang(self, line, cell)
      243             sys.stderr.flush()
      244         if args.raise_error and p.returncode!=0:
  --> 245             raise CalledProcessError(p.returncode, cell, output=out, stderr=err)
      246
      247     def _run_script(self, p, cell, to_close):


      CalledProcessError: Command 'b"awk '/Jacob/' 02/baby2008.html\n"' returned non-zero exit status 2.
```

Write a script that accepts a one or more file names as argument and returns the year and a list of names in alphabetical order with their ranking:

```
[ 2006,..., (Anthony, 9), ...]
```

Write a function to load the data for a given year from a binary file

Looking at the file with a text editor shows the structure of relevant records:

```
<h3 align="center">Popularity in 2006</h3>
...
<tr align="right"><td>9</td><td>Anthony</td><td>Sophia</td>
```

You can start from the following template, Try to make use of the *argparse* module to add features to the script (e.g. an output file name).

```
import pickle
import re

def extract_names(filename):
  """
  Given a file name for, returns a list starting with the year followed by the name and strings
  """
  # you need: year, names and a counter

  #create a regular expression to get the year
  year_match = re.search(r'Popularity\sin\s(\d+)<', text)


def save_data(namedict):
    """
    save a proper data structure
    """



def main():
  args = sys.argv[1:]

  if not args:
    print('No input data')
    quit()

  # for filename in args extract year names ...

  # save data

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

Solution: have a look at babynames3.py

11 The End!