Section 4: Control Flows and Functions

Control Flows

In a typical programming language, the major control flows include **Choice** and **Loop**.

Choice and `if` statements in Python

General form:

```
if test_1:     # test_1 should return a boolean result -- don't forget the colon: here
   statement_1 # associated block of test_1 -- don't forget the indentation here
elif test_2:   # optional, if we have multiple branches
   statement_2
else:          # optional
   statement_3
```

In [ ]:

```
x = -5

if x > 0:
    print('positive number')
elif x == 0:  # using == to test the equivalence of values. Note that keyword "is" is checking the
    print('zero')
else:
    print('negative number')
```

In [ ]:

```
x = 1
mylist = [1,2,3]

if x in mylist: # using keyword "in" to test if x is the element of list
    print('x is in the list')
else:
    print('x is not in the list')
```

In [ ]:

```
x = 10
if x > 0 or x < 0: ##  "and,or,not" are three typical boolean expressions in python
    print('non-zero number')
else:
    print('zero number')
```

In [ ]:

```
x = 10
if not x == 0: # or you can write if x!=0
    print('non-zero number')
else:
    print('zero number')
```

Remark: I highly recommend you DO NOT use the `&` and `|` in if statement -- always use `and` and `or`. In Python, `and` and `or` are logical operators, while `&` and `|` are bitwise operators that may cause unexpected problems.

Loop: `while`

```
while test:  # test returns a boolean
    statement_1
else:             # a special feature about python that is overlooked! Use it in
combination with break/continue
    statement_2
```

In [5]:

```
n = 0
mylist = [] # create an empty list
while n < 10:
    mylist.append(n) # the code to be executed if n < 10
    n = n + 1 # increase the counter by 1
    print(id(mylist))
print(mylist) # this line is no longer in the while loop!
```

```
140459607198448
140459607198448
140459607198448
140459607198448
140459607198448
140459607198448
140459607198448
140459607198448
140459607198448
140459607198448
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [6]:

```
# determine whether y is prime
y = 15
x = y // 2 # Why? Can it be improved?
while x > 1:
    if y % x == 0: # Remainder 0 -- is divisor!
        print('y is not prime')
        break      # exit the while loop immediately
    else:          # this else is for if
        x = x-1
else:              # this else is for while -- run this if only there is normal exit without hittir
    print('y is prime') # what if this statement is not in the else block?

print(x)
```

```
y is not prime
5
```

Loop: for

```
for target in object:
    statement_1
    if test_1: break # exit the for loop immediately
else:                # run this only when exit normally without hitting break
    statement_2
```

Computing sum of the list

- Iterating the list directly
- Iterating through the index

In [7]:

```
#iterating the list
mylist = [1,2,3,4]
mysum = 0

for x in mylist:
    mysum = mysum + x
```

```
print(mysum)

# this might be a more pythonic way!
```

```
10
```

In [8]:

```
#iterating through index
mylist = [1,2,3,4]
mysum = 0

for i in range(len(mylist)):
    mysum = mysum + mylist[i]
print(mysum)

# this is what you're familiar in Matlab perhaps!
```

```
10
```

By using the `enumerate()` we can actually iterate in both ways simultaneously!

In [9]:

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

for i,x in enumerate(mylist): # pay attention to the order (i,x)
    print(x)
    print(id(x))
    print(mylist[i])
    print(id(mylist[i]))
```

```
[1, 2]
140459606259184
[1, 2]
140459606259184
[3, 4]
140459606636496
[3, 4]
140459606636496
```

Something tricky: Change the elements of list

In [10]:

```
mylist = [1,2,3,4]
print(id(mylist))

for i in range(len(mylist)):
    mylist[i] = mylist[i] + 1

print(mylist)
print(id(mylist))
```

```
140459607182864
[2, 3, 4, 5]
140459607182864
```

In [11]:

```
# this will NOT work -- think why !
mylist = [1,2,3,4]

for x in mylist:
    x = x + 1
    print(x)

print(mylist)
```

```
2
3
4
```

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

A more *pythonic* way is through **list comprehension**

```
new_list = [A for B in C if D]
```

In [12]:

```
mylist = [1,2,3,4]
print(id(mylist))

mylist = [x+1 for x in mylist] #creating a new list

print(mylist)
print(id(mylist))
```

```
140459606258784
[2, 3, 4, 5]
140459606251952
```

Comprehension is very powerful -- it can also be combined with if statement to 'filter' elements. Let's see the following example.

In [13]:

```
dir(mylist)
```

Out[13]:

```
['__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 [16]:
```
# take all the special attributes/methods of mylist
dir_mylist = dir(mylist)
special_names = [name for name in dir_mylist if name.startswith('__')]
print(special_names)
```

```
['__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__', '__setatt
r__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__']
```

Note that below is the equivalent way in normal for loop -- using three lines to write if we don't use list comprehension!

In [17]:
```
special_names = []
for name in dir_mylist:
    if name.startswith('__'): # startwith() is the method for python object str
        special_names.append(name)
print(special_names)
```

```
['__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__', '__setatt
r__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__']
```

I highly recommend this video for writing the pythonic codes. Below are some more sophiscated examples -- in fact, too many loops/conditions in list comprehension can make the code less readable!

In [18]:
```
obj = ["Even" if i%2==0 else "Odd" for i in range(100)]
print(obj)
```

```
['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even',
'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'E
ven', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Od
d', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Eve
n', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Od
d', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Eve
n', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Od
d', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd']
```

In [19]:
```
vec = [[1,2,3], [4,5,6], [7,8,9]]
vec_flat = [num for elem in vec for num in elem] # nested loop
print(vec_flat)
```

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

Functions

Defining the Function

To define the function:

```
def func_name (arg1, arg2 = value):
      statements
      return value # if there's no return statement, just return None
```

Here `arg1` is the normal argument, while `arg2` has the default value if no object is passed during the call. Note here that the order is important -- normal arguments first, followed by arguments with default values.

In Python, functions are also objects! So running the codes above will create the function object, and using the name `func_name` to point to this object.

Remark: In some languages like C, there's clear distinction between terms *parameters* (when defining) and *arguments* (when calling), while in Python we can just use the words more casually.

In [20]:
```
def simple_function():
    '''this is a very simple function with neither input (arguments) nor output (return)'''
    pass # pass indicates an empty block of statements
```

In [21]:
```
y1 = simple_function() # y1 points to the return value
y2 = simple_function # y2 points to the function object, later you can just call the function by y2
print([y1,y2])
```

```
[None, <function simple_function at 0x7fbf4cfbf320>]
```

In [22]:
```
type(None)
```

Out[22]:
```
NoneType
```

In [23]:
```
dir(y2)
dir(simple_function) # return the same lists of attributes/methods of our simple_function function
```

Out[23]:
```
['__annotations__',
 '__call__',
 '__class__',
 '__closure__',
 '__code__',
 '__defaults__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__get__',
 '__getattribute__',
 '__globals__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__kwdefaults__',
 '__le__',
 '__lt__',
 '__module__',
 '__name__',
 '__ne__',
 '__new__',
 '__qualname__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__']
```

In [24]:
```
y2.__doc__
```

Out[24]:
```
'this is a very simple function with neither input (arguments) nor output (return)'
```

In [27]:
```
help(y2)
```

```
Help on function simple_function in module __main__:

simple_function()
    this is a very simple function with neither input (arguments) nor output (return)
```

By the way, `help()` is very useful for built-in types/functions or other classes/ functions defined in packages.

In [ ]:
```
help(list)
```

In [ ]:
```
help(abs)
```

Let's see what does the return statement do here.

In [28]:
```
def create_list():
    mylist = [1,2,3]
    print(id(mylist))
    return mylist  #The return statement just pass the object to output
```

In [30]:
```
output_list = create_list()
print(output_list)
print(id(output_list))
```

```
140459607330400
[1, 2, 3]
140459607330400
```

A final remark here is that whenever the `return` is executed in Python, the function will "jump out" and all the remaining statements after return will not be executed -- so be cautious when you write return in the loops! An alternative way might be you just modify some variable(name) in the loop, and return the variable at the end of your function.

In [35]:
```
def list_square(l):
    ls = []
    for x in l:
        ls.append(x**2)
    return ls
```

In [36]:
```
list_square([4,5,6])
```

Out[36]:
```
[16, 25, 36]
```

Calling the Function

When calling the functions, the arguments can be matched by position (normal argument) or by name (key word argument). We will omit the more complicated cases in our course. Let's learn through this simple example.

In [37]:
```
def func(a, b, c=3, d=4):
    print([a, b, c, d])
```

In [38]:
```
func(1, 2) # a=1, b=2

func(1, 2, 3, 4) # a=1, b=2, c=3, d=4

func(1, c=0, b=0) # a=1, b=0, c=0, d=4
```

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

Argument Passing: Passing by *Object Reference* in Python

In Matlab, the arguments are usually **passed by value** in the functions. However, Python functions **pass the arguments by object reference**.

For simplicity, below we do not discuss the global variables here (As the famous saying goes, *global variables are evil in object-oriented languages*).

In Python, suppose we have an object named `obj_python` that is passed to a function `func(obj_func)`. What does the function do is create a new name (identifier) by `obj_func = obj_python` that points to the **same object** (instead of creating a new object!). All the statements within function are then executed with the name (identifier) `obj_fun`, and the name `obj_fun` will be destroyed after calling the function.

- For *mutable objects*, the modification with `obj_fun` inside the function may change the value of object, which is pointed by the name `obj_python` outside the function.
- For *immutable objects*, since the value cannot be modified once the object is created, the function will not affect the object pointed by `obj_python`.
- That's why some people say in Python, the immutable objects are passed by values, while the mutable objects are passed by reference or pointer -- they are indeed the "net effects". In fact, these observations are the reflections of **passing by object reference** mechanism in Python.

In [39]:
```
def modify_list(mylist):
    '''modify the first element of list '''
    print(id(mylist))
    mylist[0] = 100 # Note here we don't return anything (or return None)
    y = mylist[0] # this y is just local name -- will be destryed if we don't return (passing it to
```

In [43]:
```
mylist = [1,2,3]
print(id(mylist))

y = modify_list(mylist = mylist) #by calling the function, we have another results printed.

# Note the left mylist just means the keyword in function, and the right keyword means
# the argument object we are passing into the function!

print(y)
print(mylist)
print(id(mylist))
```

```
140459606493296
140459606493296
None
[100, 2, 3]
140459606493296
```

Of course, it might be a better practice to avoid the same parameter and input object name.

In [44]:
```
mylist1 = [1,2,3]
modify_list(mylist = mylist1)
print(mylist1)
```

```
140459606624448
[100, 2, 3]
```

This reminds us about the `reverse()` or `sort()` methods of `list` that we talked about in the last lecture --

modifying the mutable ojects **in place**.

Compare with other examples:

In [ ]:
```
def modify_list_complete(mylist):
    '''modify the list completely by creating a new one, without return'''
    mylist = [100,2,3] # this mylist is just local name -- will be destryed if we don't return (pas
```

In [ ]:
```
mylist = [1,2,3]
modify_list_complete(mylist = mylist)
print(mylist)
```

In [ ]:
```
def modify_list_complete_new(mylist):
    '''modify the list completely by creating a new one, and return'''
    mylist = [100,2,3]
    return mylist
```

In [ ]:
```
mylist = [1,2,3]
y = modify_list_complete_new(mylist = mylist)
print(mylist)
print(y)
```

Now use the following example to test if you really understand :

In [47]:
```
def modifier(myint, mylist):
    '''modify the immutable integer and mutable list simultaneously'''
    myint = 1000
    mylist[0] = 1000


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

y = modifier(a,b)
print(a)
print(b)

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

modifier(a,b.copy())
print(a)
print(b)
```

```
1
[1000, 2, 3]
1
[1, 2, 3]
```

Take-home message: Don't change mutable arguments in Python functions unless you intend to do it. This is something really different with Matlab!

Lambda Function

Lambda function provides a convenient way for defining simple functions. Despite its simplicity, Guido Van Rossum used to consider remove it in Python 3.

In [50]:
```
f_square = lambda x: x*x

mylist = list(range(10))
```

```
mylist_square = [f_square(x) for x in mylist]
print(mylist_square)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```