Introduction to Programming Lecture 3

- Instructor : Andrew Yarmola andrew.yarmola@uni.lu
- Course Schedule : Wednesday 14h00 - 15h30 Campus Kirchberg B21
- Course Website : sites.google.com/site/andrewyarmola/itp-uni-lux
- Office Hours : Thursday 16h00 - 17h00 Campus Kirchberg G103 and by appointment.

Remarks on homework and questions

Permutations

This appears to have been the most challenging question. Recall that a permutation is a bijection $f : \{0, \dots, n-1\} \to \{0, \dots, n-1\}$. To a python tuple $a = (a_0, \dots, a_{n-1})$ we a assign the permutation $f_a(i) = a[i] = a_i$.

Here is my take on how to do the homework problem.

In [1]:

```
def is_perm(p) :
    # go through the numbers x = 0, ..., len(p)-1
    for x in range(len(p)) :
        # p is permutation if and only if each
        # x appears in p (i.e. f_p is surjective)
        if x not in p :
            return False
    return True

def compose(a,b) :
    # If a or b are not permutations or if they are
    # permutations of different length, we return the empty tuple
    if not is_perm(a) or not is_perm(b) or len(a) != len(b) :
        return ()
    # we can compose two permtuations as if they were functions
    result = [ b[a[i]] for i in range(len(a)) ]

    # we must return a tuple!
    return tuple(result)

# Remark :
# List comprehension is the same thing as a for loop. The code :
#
# result = [ b[a[i]] for i in range(len(a)) ]
#
# is the same thing as the code
#
# result = []
# for i in range(len(a)) :
#     result.append(b[a[i]])
#
```

Notice that I reuse my previous functions! I do not rewrite `is_perm` inside of `compose`, I just call it.

Remarks on input

Notice that `is_perm` takes **one** argument, which is assumed to be a tuple. So when I call the function, I should give it one argument as a tuple. The second function, `compose` takes **two** arguments where each argument is a tuple.

In [2]:

```
is_perm((1,2,0,3))
```

Out[2]:

```
True
```

In [3]:

```
is_perm(1,2,0,3) # this is giving the function 4 arguments!
```

```
---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-3-bf8cb62a9e97> in <module>()
----> 1 is_perm(1,2,0,3) # this is giving the function 4 arguments
!

TypeError: is_perm() takes 1 positional argument but 4 were given
```

Remarks on output

The output of your function should be a **consistent** type. It is **not** advisable to return error strings if there is a problem with the input.

In [4]:

```
def is_square(n) :
    if type(n) is not int :
        # Do not do this! Clearly the function should only
        # return a boolean value!
        return "Error, need to provide integer"
    else :
        for x in range(n) :
            if x**2 == n :
                return True
            elif x**2 > n :
                return False
    return False # Why do I need this here?
```

Also, if your function is supposed to return something, **don't** `print` it, `return` it! When you call `return` you actually provide the output object, it you just print the answer, you can't pass it to the rest of your code.

In [5]:

```
six_is_a_square = is_square(6)
if not six_is_a_square :
    print("6 is not a square")
```

```
6 is not a square
```

Be lazy

Don't re-implement if you can re-use (or make use of). For example, many of you wrote the `bits_needed` function from scratch. That's great. However, you can be lazy!

In [6]:

```
def bits_needed(n) :
    if n == 0 :
        return 1
    else :
        return len('{0:+b}'.format(n))
```

In [7]:

```
# Recall that b is the format for binary and
# the + means always print a sign
'{0:+b}'.format(-10)
```

Out[7]:

```
'-1010'
```

You can also see if there is some built-in function that gives you binary representations. As it turns out, there is!

In [8]:

```
help(bin)
```

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

bin(number, /)
    Return the binary representation of an integer.

    >>> bin(2796202)
    '0b1010101010101010101010'
```

In [9]:

```
bin(-10)
```

Out[9]:

```
'-0b1010'
```

In [10]:

```
def bits_needed_v2(n) :
    bin_str_len = len(bin(n))
    if n > 0 :
        return bin_str_len - 1
    else :
        return bin_str_len - 2
```

In [11]:

```
print(bits_needed_v2(0))
print(bits_needed_v2(-10))
print(bits_needed(-10))
```

```
1
5
5
```

Docstrings, comments, and writing style

Along with **good variable names**, docstrings and comments are a great way to keep your code readable and organized. Let's begin with comments.

Python allows you to comment out a whole line or the **end** of a line. To start a comment, use just use the # symbol. Everything after that symbol, to the end of the line, is a comment for the reader and will be ignored by the interpreter.

In [12]:

```
# This is a comment

a = 4 # This is a comment on the same line as some code

# To make multi-line commnets
# just place a # at the beginning of every line
```

You should keep your **line length** around **80** characters for readability. **In general, the end of a line in python means the end of a statement or command**. However, sometimes you have a very long expression and you want to use multiple lines to make it look clean.

If your line is getting too long, you can use \ symbol to tell python to **explicitly** continue on the next line. Additionally, inside `[]`, `()` and similar brackets, python will **implicitly** continue reading on the next line.

In [13]:

```
# Continuing a line explicitly using \
if 6 < 4 \
    or 2 in [0,2,3] \
    and 'something' != 'else' :
        print("That's a lot of conditions to check.")

# You can also do this in declarations
a = 'this ' + 'is ' + 'not ' + 'the ' + 'best ' +\
    'example ' + 'however ' + 'I ' + 'hope ' + 'you ' + 'get ' +\
    'the ' + 'point'

print(a)
```

```
That's a lot of conditions to check.
this is not the best example however I hope you get the point
```

Remark : Notice that in the above `if` statement, the `or` operator is evaluated **before** the `and`. If you want to be explicit, use `()` around your `and`, `or` expressions.

Warning : you cannot put *any* characters after the line continuation character \. Not even a space! Not even a comment.

In most cases, you should use implicit continuation. It generally reads better.

In [14]:

```
# For continuing implicitly, we can use ()
if (6 < 4
    or 2 in [0,2,3]
    and 'something' != 'else') :
        print("That's a lot of conditions to check.")

# In declaratinons we can also use implicit continuation
a = ['a', 'very', 'very', 'very', 'very', 'very',
    'very', 'very', 'long', 'list']

print(a)
```

```
That's a lot of conditions to check.
['a', 'very', 'very', 'very', 'very', 'very', 'very', 'very', 'lon
g', 'list']
```

Docstrings

It is important to document your functions so that users will know what kind of input the function expects and what kind of output it will produce. To documents a function, we use **docstrings** in python, which are just **triple-quoted** strings on the line right after the `def` statement.

In [15]:

```
def bits_needed(n) :
    """
    Computes the number of bits needed to represent
    an integer. The count includes the sign.

    Parameters
    ----------
    n : int, required
    """
    return len('{0:+b}'.format(n))
```

In [16]:

```
help(bits_needed)
```

```
Help on function bits_needed in module __main__:

bits_needed(n)
    Computes the number of bits needed to represent
    an integer. The count includes the sign.

    Parameters
    ----------
    n : int, required
```

Box problem

Here is a fun probability problem. A game show has a team of 10 contestants. The contestants are numbered $0, \ldots, 9$.

In the game room, there are 10 boxes, also numbered $0, \ldots, 9$. Each box contains one of the numbers $0, \ldots, 9$. The game show host makes the following bet with the contestants

- each contestant pays €10 to play
- the numbers inside the boxes are randomly shuffled **once** for the entire game.
- one by one, contestants enters the room and are given 5 chances to find their number (i.e. they get to open any 5 boxes they choose, one at a time)
- after a contestant has either found their number or opened 5 boxes, the room is reset just as it was before the contestant went inside
- if all the contestants find their number, the game show will award each €300, however, if any contestant has failed to find their number, the game show keeps all the money

Players can decide on a strategy ahead of time, but cannot communicate once the game begins. Would you play this game?

We can use a list on the numbers $0, \ldots, 9$ to encode the shuffled boxes

In [17]:

```
import random

boxes = list(range(10))
# Shuffle the elements randomly in place
random.shuffle(boxes)

print(boxes)
print('Box 0 contains {}, box 1 contains {}, etc.'.
      format(boxes[0], boxes[1]))
```

```
[7, 6, 2, 1, 0, 3, 5, 8, 4, 9]
Box 0 contains 7, box 1 contains 6, etc.
```

In [18]:

```
print(random.sample(range(10),3))
```

```
[0, 7, 4]
```

In [19]:

```
def bad_strategy(boxes, person) :
    """ Given the boxes in the room and the person's number.
    Try to find the number by randomly choosing 5 boxes"""
    # Pick 5 random boxes
    places_to_look = random.sample(range(10),5)
    for place in places_to_look :
        if boxes[place] == person :
            return True
    return False
```

In [20]:

```
one_person_results = []
while len(one_person_results) < 10001 :
    one_person_results.append(bad_strategy(boxes,1))
sum(one_person_results)/len(one_person_results)
```

Out[20]:

```
0.502049795020498
```

Let us see how well the `bad_strategy` does. We will shuffle the boxes many times and see if all ten people can win.

In [21]:

```
import random

def play_box_game(strategy, num_runs) :
    """Given a strategy and num_runs, will play the
    game num_runs times. For each run we randomly shuffle
    the box. Returns the list of game outcomes."""
    boxes = list(range(10))
    game_results = []
    while len(game_results) < num_runs + 1 :
        result = True
        # Shuffle the boxes once per game
        random.shuffle(boxes)
        for person in range(10) :
            result = result and strategy(boxes,person)
            if result is False : # We have lost !!!
                # So we stop trying !!! Call break
                # to exit the person for-loop,
                # but NOT the outer while loop
                break
        game_results.append(result)
    return game_results
```

In [22]:

```
game_results = play_box_game(bad_strategy, 100000)
print(sum(game_results)/len(game_results))
```

```
0.0007999920000799992
```

In [23]:

```
## This is what we expect our probablity of winning to be
0.5**10
```

Out[23]:

```
0.0009765625
```

I happen to have a secret strategy that each player can use so that we do much better. Let's test out how much better.

In [24]:

```
# import the module that contains my secret strategy
import secret_strategy

game_results = play_box_game(secret_strategy.good, 100000)

print(sum(game_results)/len(game_results))
```

```
0.35419645803541966
```

On the homework, you will try to come up with your own strategy and see if you can beat mine!

Objects

Python is built around objects. In fact, "everything is an object" is one of the key mantras of the language. We will think of an **object** in python as a structure that contains *data* (i.e. values we are interested in) and *functions* (i.e. computer code). These are called the **attributes** of an object and are accessed using the `object.attribute_name` notation. Attributes can, therefore, either be **data attributes** or **method attributes**.

In [25]:

```
a = 2.5 + 5.6j
a.imag # A data attribute that returns a float
```

Out[25]:

```
5.6
```

In [26]:

```
a.conjugate # a method attribute
```

Out[26]:

```
<function complex.conjugate>
```

Above, `a.conjugate` returns the **function** object that contains the code to run `complex.conjugate`. To run a function, we use the `a.conjugate()` syntax. This returns the conjugate of `a`.

Up to this point we have been working with instance objects. An **instance object** is one that conforms to some recipe. The object itself is a particular preparation of that recipe. For example,

In [27]:

```
a = float(1.5) # an instance object conforming to the float recipe
b = 5.6 # another instance object conforming to the float recipe
c = float # the object that describes the float recipe
print(a, b, c)
```

```
1.5 5.6 <class 'float'>
```

One of the benefits of python's approach is that objects can always be

- assigned a variable name (or several)
- added to a list (and most collection objects)
- passed as an argument to a function

Functions are objects !

This makes is very easy to pass functions around. In the box problem, you already saw that I can pass the `strategy` function to my `play_box_game` code.

For another example, you could make a custom function to apply to every element of a list.

In [28]:

```
def custom(data) :
    return 3*data

a = [1,2,3,4]
custom_a = list(map(custom, a)) # apply custom to each element in a
print(custom_a)
```

```
[3, 6, 9, 12]
```

Above, you see the wonderful `map` function.

- `map(func, list_1, list_2, ...)` applies `func` to the elements of the supplied lists. When multiple lists are supplied, map returns an iterator object containing `func(list_1[i], list_2[i], ...)`, until the shortest list is exhausted.

Here is another example.

In [29]:

```
def diff_sq(a,b) :
    """ Return the difference squared """
    return (a - b)**2

def dist(x,y) :
    """
    Return the Euclidean distance between vectors x and y.

    Implementation detail
    ---------------------
    If x is shorter than y (or vice versa) then y is truncated
    to the length of x """
    # Here, we map the diff_sq function over the two lists, sum the results,
    # and take the square root
    return sum(map(diff_sq, x, y))**(1/2)

print(dist((1,0,2),(2,0,7)))
```

```
5.0990195135927845
```

I stated that `map` returns an **iterator**. An **iterator** is an object that represents a steam of data. Basically, it's an object that supports a `.__next__()` method or can be passed as an argument to the `next(some_iterator)` function. This is a convenient construct because it allows for **lazy evaluation**, i.e. the value of the next object is only computed as needed. For example,

In [30]:

```
import random

# generate some random points in the plane with both coordinates
# between 0 and 1
# note : there are better ways to do this, we as will learn later
x_points = [ (random.random(),random.random()) for i in range(10) ]
y_points = [ (random.random(),random.random()) for i in range(10) ]

# print the first distance that's closer than 1/3
idx = 0
map_iterator = map(dist, x_points, y_points)
for d in map_iterator :
    if d < 1/3 :
        print("""Samples {} and {} were at index {}
        and have distance {} < 1/3.""".format(x_points[idx],
        y_points[idx], idx, d))
        break
    idx += 1
```

```
Samples (0.5351955505225942, 0.3904588822033368) and (0.7624927502
89669, 0.44733699043912767) were at index 1
        and have distance 0.23430564700500897 < 1/3.
```

Above, the `dist` function was never called for point after the found index.

Functions, passing by value, and scope

In python, parameters are passed to functions as references to objects in memory. Therefore, you can modify the input value of a function whenever the object is mutable. For example,

In [31]:

```
def reverse_list(L) :
    """Sorts the contents of L in reverse order. Returns L."""
    print("Local id of L is {}.".format(id(L)))
    L.sort(reverse=True)
    return L
```

In [32]:

```
import random
# pick 5 random numbers from 0,...,9
rand_list = random.sample(range(10),5)
print(rand_list)
```

```
[6, 1, 7, 4, 8]
```

In [33]:

```
print("Global id of rand_list is {}".format(id(rand_list)))
```

```
Global id of rand_list is 4442117192
```

In [34]:

```
reversed_list = reverse_list(rand_list)
print("Global id of rand_list is {}".format(id(reversed_list)))
```

```
Local id of L is 4442117192.
Global id of rand_list is 4442117192
```

In [35]:

```
rand_list is reversed_list
```

Out[35]:

```
True
```

As you can see, `rand_list`, `reverse_list` and even `L` (while inside the function) point to the same bit of memory.

Python does does following when you call `reverse_list`

- declare a namespace for the function call of `reverse_list` (this is called a **stack frame**)
- copy the reference from `rand_list` to the **local** name `L`
- run the indented code under the `def` line
- the local namespace is cleared (i.e. the names used locally in the function are forgotten)

The **scope** of a variable name is a *textual* region of a python program where that variable name is directly accessible.

In [36]:

```
L # Even though we ran the function above,
# L is not variable name outside of the function call
```

```
---------------------------------------------------------------------
---------
NameError                                 Traceback (most recent c
all last)
<ipython-input-36-a635d6a94e2e> in <module>()
----> 1 L # Even though we ran the function above,
      2 # L is not variable name outside of the function call

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

Above, `L` has only **local** scope in the function `reverse_list`. Namespaces are searched list a nested set. For example,

In [37]:

```
x = 5
def add_x(y) :
    return x+y # x does not exist in the local scope,
               # so it is pulled from the next one up
```

In [38]:

```
add_x(6)
```

Out[38]:

```
11
```

Here, `x` has `global` (or `module`) level scope and can only be found is the global namespace.

In [39]:

```
x = 5
def add_7(y) :
    x = 7 # x is declared in the local scope
    return x+y # x now has local scope,
               # it is first found in the local namespace
```

In [40]:

```
add_7(7) # inside the function x = 7
print(x) # outside the function x = 5
```

```
5
```

Global variables

It is not recommended, but at times your code might work better (or read better) if you have a global variable that you would like to reassign or modify in the body of a function.

In [41]:

```
L = 'a global list'.split() # outside we have L
print("Global id of L is {}".format(id(L)))
reverse_list(rand_list) # inside we also have L
```

```
Global id of L is 4442118792
Local id of L is 4442117192.
```

Out[41]:

```
[8, 7, 6, 4, 1]
```

In [42]:

```
def reverse_global() :
    global L # we can declare that we us the global L
    L.sort(reverse=True)
    return L
```

In [43]:

```
reverse_global()
```

Out[43]:

```
['list', 'global', 'a']
```

In [44]:

```
# cannot have both a local and global L
def reverse_global_with_arg(L) :
    global L
    L.sort(reverse=True)
    return L
```

```
  File "<ipython-input-44-f551a8367310>", line 3
    global L
    ^
SyntaxError: name 'L' is parameter and global
```

Positional, named and default parameters

So far, for functions with multiple arguments we have used **position** to specify which parameter is set to which local variable. For example,

In [45]:

```
def print_args(a,b) :
    print(8*'-')
    print("a is", a)
    print("b is", b)
    print(8*'-')

print_args(5,'foo')
print_args('bar',6)
```

```
--------
a is 5
b is foo
--------
--------
a is bar
b is 6
--------
```

As you may have noticed earlier, I used a **named parameter** `reverse` in the method call `sort(reverse = True)`. Python allows you to specify the parameter name in the function call

In [46]:

```
print_args(b = 5, a = 'foo')
```

```
--------
a is foo
b is 5
--------
```

In [47]:

```
# What happens if you don't give all the parameters?
print_args(b = 5)
```

```
---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-47-8e26397347c1> in <module>()
      1 # What happens if you don't give all the parameters?
----> 2 print_args(b = 5)

TypeError: print_args() missing 1 required positional argument: 'a
'
```

In [49]:

```
# What happens if you make up a variable
# name at the time of the call?
print_args(b = 5, c = 'bar')
```

```
---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-49-ede34a0f1774> in <module>()
      1 # What happens if you make up a variable
      2 # name at the time of the call?
----> 3 print_args(b = 5, c = 'bar')

TypeError: print_args() got an unexpected keyword argument 'c'
```

In fact, the parameter `reverse` in the function call `sort(reverse = True)` also an **optional parameter**. These are also known as **keyword** parameters. Optional parameter can be specified as part of the function definition. They must come **after** positional argumetns and they need to be given a **function-global default value**.

In [50]:

```
# we define optional parameters the_list, reverse, and print_list
def append_and_sort(data, the_list = [],
                    reverse = False, print_list = False) :
    the_list.append(data)
    # how can we use reverse = revese? Explain!
    the_list.sort(reverse = reverse)
    if print_list :
        print('The list is', the_list)
```

In [51]:

```
L = [3,2]
append_and_sort(5,L)
print(L)
```

```
[2, 3, 5]
```

Above, we can still use position to specify parameters. However, we did not need to specify the value of `reverse`, which defaults to `False`.

We can continue to use position to specify the values of optional paramters. However, for optional parameters that have boolean values **this is not recommened.**

In [52]:

```
# Which arguemnt is set to True? Hard to tell.
# This is BAD code
append_and_sort(5, L, True)
print(L)
```

```
[5, 5, 3, 2]
```

It is **much better** to use the named parameter format (also called keyword format).

In [53]:

```
# A mix of positional and optional
append_and_sort(5, L, print_list = True)
```

```
The list is [2, 3, 5, 5, 5]
```

Above, we wanted to use the default value of `reverse` but a different value for `print_list`. We **must** use named parameters if we want to *skip* an optional parameter while using some positional arguments.

Warning about default values

The default value of an optional parameter is a **function-global value**. This means that when the interpreter reads the definition of a function and creates the function object, it constructs the values of the default parameters **for all future function calls**.

Recall that a **function object** is one that store the **computer code** of the function. In addition to this, it also stores the default values.

In [54]:

```
append_and_sort(3,print_list = True)
append_and_sort(3,print_list = True)
append_and_sort(3,print_list = True)
```

```
The list is [3]
The list is [3, 3]
The list is [3, 3, 3]
```

Above, the default value of `the_list` is created **once** when the function definition is first read by the interpreter. Thus, **every time that the default value is used, it's the same piece of memory**! Keep in mind that there are two stages of memory allocation for functions.

- Function object stage : the interpreter reads the function definition and creates a function object. The default values of the function object are set.
- Function is called : the interpreter creates a new stack frame and copies the references to the function inputs. If default values are not reassigned by the input, those from the object creation stage are used.