# Introduction to Programming Lecture 9

- 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

For Exercise 2b on Homework 8, I forgot to list `__rsub__` as something you should implement.

## A few remarks

### Function annotation

If you ever encoutner something like

In [1]:

```
def add(x : int, y : int) -> int:
    return x + y
```

know that this is simply **function annotation**. It is simply for **readabily** and **has no impact on how the code runs.** Python will **not** check the types for you if you write code this way. The above is **equivalent** to

In [2]:

```
def add(x,y) :
    return x + y
```

### Property deletion

When we talked about the `@property` decorator, we implemented getters and setters. One other property operation you might want to do is **deletion**. For example, you might have an object that creates a very large amount of data that you only use once or twice but it becomes invalid after you change something or you just want to delete it. We can use the `@property_name.deleter` syntax. Here is an example :

In [3]:

```
import random

class RandomData :
    def __init__(self) :
        self._size = 0
    @property
    def data_size(self) :
        return self._size
    @property
    def data(self) :
        if hasattr(self, '_data') is False :
            print("Generating data.")
            self._data = random.sample(range(1000000),100000)
            self._size = 100000
        return tuple(self._data)

    @data.deleter
    def data(self) :
        # do any cleanup
        print("Cleaning up.")
        self._size = 0
        del self._data
```

In [4]:

```
a = RandomData()
print(a.data_size)

print(len(a.data))
print(len(a.data))

print(a.data_size)

del a.data

print(a.data_size)

print(len(a.data))

del a.data
```

```
0
Generating data.
100000
100000
100000
Cleaning up.
0
Generating data.
100000
Cleaning up.
```

**Remark** If you don't declare a setter or a deleter, python will not let a user set or delete a property using the nice property syntax. **You don't have to declare a deleter (or setter) unless you think a user will want it.**

## Requiring keyword arguments without defaults

It might happen that you want to require some keyword arguments in your code, here is the simple way to do that

In [5]:

```
def do_stuff(arg1, arg2, *, required_keyword) :
    print(arg1, arg2, required_keyword)
```

In [6]:

```
do_stuff(1,2)
```

```
---------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-6-336a68e59b02> in <module>()
----> 1 do_stuff(1,2)

TypeError: do_stuff() missing 1 required keyword-only argument: 'r
equired_keyword'
```

In [7]:

```
do_stuff(1,2, required_keyword = None)
```

```
1 2 None
```

In [8]:

```
# the * doesn't eat extra positional arguments
do_stuff(1, 2, 4, required_keyword = None)
```

```
---------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-8-d710f5cefb17> in <module>()
      1 # the * doesn't eat extra positional arguments
----> 2 do_stuff(1, 2, 4, required_keyword = None)

TypeError: do_stuff() takes 2 positional arguments but 3 positiona
l arguments (and 1 keyword-only argument) were given
```

In [9]:

```
def do_stuff_v2(arg1, arg2, *rest, required_keyword) :
    print(arg1, arg2, required_keyword, rest)
```

In [10]:

```
# a * followed by a varaible name, does
do_stuff_v2(1, 2, 4, 4, 5, 6)
```

```
---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-10-9deba46a8357> in <module>()
      1 # a * followed by a varaible name, does
----> 2 do_stuff_v2(1, 2, 4, 4, 5, 6)

TypeError: do_stuff_v2() missing 1 required keyword-only argument:
'required_keyword'
```

In [11]:

```
do_stuff_v2(1, 2, 4, 4, 5, 6, required_keyword = None)
```

```
1 2 None (4, 4, 5, 6)
```

## The idea of a callback

The notion of a **callback** is a very useful tool in my programming problems. A **callback** is simply a function that you call once a process (i.e. another function) has finished. Here is a simple example :

In [12]:

```
class Aggregator :
    def __init__(self) :
        self._count = 0
        self._total = 0
    @property
    def count(self) :
        return self._count
    @property
    def total(self) :
        return self._total
    def incriment(self, val) :
        print("Incrimenting total and count")
        self._count += 1
        self._total += val

def some_function(n, *, callback) :
    # do stuff
    result = 1/n
    callback(result)
```

In [13]:

```
agr = Aggregator()

some_function(4, callback = agr.incriment)
some_function(6, callback = agr.incriment)
some_function(7, callback = agr.incriment)

print("Have {} total and called {} times". format(agr.total, agr.count))
```

```
Incrimenting total and count
Incrimenting total and count
Incrimenting total and count
Have 0.5595238095238095 total and called 3 times
```

As you can see above, I was able to store counters and see how many times my callback as been used. This can be very useful if your code doesn't admit a nice `for` loop for you to keep track of things, or you have a parallelized comptuation.

# Functional programming

As we have seen, functions are objects in python. Thus, it is only natural that there would be many tools for manipulating functions and applying them in clever ways. We have already seen tools such as `map`. Here are a few more.

## `lambda` and anonymous function

Python allows for the creation of (**simple**) functions using $\lambda$-calculus notation. Let's start with an example :

In [14]:

```
add = lambda x,y : x + y
# add is now a function that takes two inputs and
# returns their sum
```

In [15]:

```
add(8,2)
```

Out[15]:

```
10
```

The `lambda` keyword is followed by the list of arguemnts, then a `:` separates the arguments from the result. This can be very usefull if you want to sort by tuples of numbers by their sum.

In [16]:

```
from random import randint

r = (-10,10)

data = [(randint(*r), randint(*r), randint(*r))
        for _ in range(5) ]

print(data)

# sort will now use the *sum* of a tuple
# to compare the tuples
data.sort(key = lambda x : sum(x))

print(data)
```

```
[(-7, -3, -1), (-6, -5, 0), (-2, 7, 9), (1, -2, 7), (8, -8, -1)]
[(-7, -3, -1), (-6, -5, 0), (8, -8, -1), (1, -2, 7), (-2, 7, 9)]
```

In [17]:

```
names = ['David Zetto', 'Brian Bates',
         'Raymond brower', 'Ned Andrews']
```

In [18]:

```
# sort by last name lowercase
print(sorted(names, key=lambda name: name.split()[-1].lower()))
```

```
['Ned Andrews', 'Brian Bates', 'Raymond brower', 'David Zetto']
```

**Remark** Sometimes `lamba` expressions get hard to read and it's just cleaner to use `def`. It is **up to you** which you prefer.

In [19]:

```
def some_func() :
    names = ['David Zetto', 'Brian Bates',
         'Raymond brower', 'Ned Andrews']
    def lower_lastname(name) :
        return name.split()[-1].lower()
    print(sorted(names, key=lower_lastname))

some_func()
```

```
['Ned Andrews', 'Brian Bates', 'Raymond brower', 'David Zetto']
```

## the `operator` module

For commonly used functions such as `+`,`-`,, .etc, getting an item by key or index, or using an attribute of an object, the `operator` module provides manyof these. Here are a few examples

In [20]:

```
from operator import itemgetter

# sort by last elements
data.sort(key = itemgetter(-1))
print(data)

data.sort(key = lambda x : x[-1])
print(data)
```

```
[(-7, -3, -1), (8, -8, -1), (-6, -5, 0), (1, -2, 7), (-2, 7, 9)]
[(-7, -3, -1), (8, -8, -1), (-6, -5, 0), (1, -2, 7), (-2, 7, 9)]
```

In [21]:

```
from operator import add

a = map(add, range(1,4), range(2,5))
print(list(a))
```

```
[3, 5, 7]
```

In [22]:

```
from operator import attrgetter

r = (-10,10)
c_nums = [ complex(randint(*r), randint(*r)) for _ in range(5) ]
a = map(attrgetter('real'), c_nums)
print(list(a))
a = map(lambda z : z.real, c_nums)
print(list(a))
```

```
[5.0, 9.0, 1.0, -3.0, 2.0]
[5.0, 9.0, 1.0, -3.0, 2.0]
```

In [23]:

```
from string import ascii_lowercase
from random import sample, randint
from operator import methodcaller

random_words = [''.join(sample(ascii_lowercase, randint(5,10)))
                for _ in range(5)]

random_words.sort(key = methodcaller('__len__'))

print(random_words)

random_words.sort(key = lambda s : len(s))

print(random_words)
```

```
['ycvnfr', 'zvtucr', 'irzmcyu', 'tyzpgqsej', 'hxqpvmcrdy']
['ycvnfr', 'zvtucr', 'irzmcyu', 'tyzpgqsej', 'hxqpvmcrdy']
```

Note, if `g = methodcaller(method_name, arga1, arg2, ...)` then `g(x) = x.method_name(arg1, arg2,...)`

In [24]:

```
random_words.sort(key = methodcaller('count', 'e'))
print(random_words)
```

```
['ycvnfr', 'zvtucr', 'irzmcyu', 'hxqpvmcrdy', 'tyzpgqsej']
```

Many functions in the `operator` module are interchangeble with good use of `lambda` notation. You can use whichever you like best.

**WARNING** In lambda functions, the variables you see are evaluated at execution time.

In [25]:

```
n = 4
a = lambda x : x + n
n = 5
print(a(3))
```

```
8
```

In [26]:

```
n = 4
# you can use keyword notation
# to force a lambda function to
# pull the local varaible at the time
# of definition
a = lambda x, y = n: x + y
n = 5
print(a(3))
```

```
7
```

## `partial` from `functools` module

If you have a function that takes many arguments, but you would like to make a version of it with different default arguments, you can use the `partial` method from the `functools` module.

In [27]:

```
def error(error_type, error_message) :
    print(error_type.__name__,'Message:', error_message)
```

In [28]:

```
from functools import partial

type_error = partial(error, TypeError)
all_fails = partial(error, error_message = "Everything is broken!")
```

In [29]:

```
type_error("Just a type error")
```

```
TypeError Message: Just a type error
```

In [30]:

```
all_fails(RuntimeError)
all_fails(TypeError)
```

```
RuntimeError Message: Everything is broken!
TypeError Message: Everything is broken!
```

Another usefull example is a logger (or a logger callback). You can write a generic logging function, and pass a partial to your exection as a callback

In [31]:

```
def log(data, log_file) :
    with open(log_file,'a') as fp :
        fp.write(data)
```

In [32]:

```
def set_username(person, name,*, callback) :
    person['username'] = name
    return callback("Set username to {}\n".format(name))

person = {}
username_log = partial(log, log_file = 'usernames.log')
set_username(person, 'john', callback = username_log)
```

In [33]:

```
%cat usernames.log
```

```
Set username to john
```

## filter

If you ever have a sequence (or iterable) and you don't some parts of it, you can use the `filter` call.

In [34]:

```
for x in filter(lambda x : x % 3 == 0, range(-10,10)) :
    print(x)
```

```
-9
-6
-3
0
3
6
9
```

As you can see, `filter` only provided us with the sequence for which the anonymous function `lambda x : x % 3 == 0` returned True.

## fun with iterators

We have seem the `map` function as a useful tool to work with iterators. Here are a few :

- `zip(iterable1, iterable2, ...)` - returns a zip object whose `.__next__()` method returns a tuple where the $i^{\text{th}}$ element comes from the $i^{\text{th}}$ iterable argument. Continues until the shortest iterable in the argument sequence is exhausted.

In [35]:

```
# zip
iterable1 = range(5)
iterable2 = range(3,10)
iterable3 = range(7,100)



for x in zip(iterable1, iterable2, iterable3) :
    print(x)
```

```
(0, 3, 7)
(1, 4, 8)
(2, 5, 9)
(3, 6, 10)
(4, 7, 11)
```

In [36]:

```
for x in zip(range(4), range(1,5)) :
    print(add(*x))
```

```
1
3
5
7
```

You can actually use `zip` with a repeated argument, you just have to be careful. Recall the distrinction : an **iterator** is an object that responds to a `__next__` method and will be exhausted after **one** use. An **iterable** is an object that can **procude** an interator for its contents.

In [37]:

```
iter1 = iter((0,1,2,3,4,5))
print(type(iter1))

for x in zip(iter1, iter1) :
    print(x)
```

```
<class 'tuple_iterator'>
(0, 1)
(2, 3)
(4, 5)
```

In [38]:

```
iterable1 = range(6)

print(type(iterable1))

for x in zip(iterable1, iterable1) :
    print(x)
```

```
<class 'range'>
(0, 0)
(1, 1)
(2, 2)
(3, 3)
(4, 4)
(5, 5)
```

Above, the `range` object produces a **new** iterator of its contents **twice** inside the call to `zip`. While the `tuple_iterator` is **used** in both arguments.

- any(iterable) - returns `True` if `bool(val) == True` for any `val` in `iterable`
- all(iterable) - returns `True` if `bool(val) == True` for all `val` in `iterable`

In [39]:

```
def has_char(char, string) :
    return any(map(lambda c : c == char, string))
```

In [40]:

```
print(has_char('e', 'Hello'))
print(has_char('f', 'Hello'))
```

```
True
False
```