Introduction to Programming Lecture 2

- 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 floating point

For the Bailey–Borwein–Plouffe formula

$$\pi = \sum_{k=0}^{\infty} \left[\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right],$$

it was likely that you would have obtained the following error if you tried to use too many values of k.

In [1]:

```
sum([ (4/(8*k+1)-2/(8*k+4)-1/(8*k+5)-1/(8*k+6))/16**k for k in range(0,300) ])
```

```
---------------------------------------------------------------------
---------
OverflowError                                Traceback (most recent c
all last)
<ipython-input-1-6afcf291a095> in <module>()
----> 1 sum([ (4/(8*k+1)-2/(8*k+4)-1/(8*k+5)-1/(8*k+6))/16**k for
k in range(0,300) ])

<ipython-input-1-6afcf291a095> in <listcomp>(.0)
----> 1 sum([ (4/(8*k+1)-2/(8*k+4)-1/(8*k+5)-1/(8*k+6))/16**k for
k in range(0,300) ])

OverflowError: int too large to convert to float
```

In [2]:

```
[ (1/16**k)*(4/(8*k+1)-2/(8*k+4)-1/(8*k-5)-1/(8*k+6)) for k in range(260,270)
]
```

Out[2]:

```
[9.7257e-320, 6.033e-321, 3.75e-322, 2.5e-323, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0]
```

Sow what is going on? Why are we getting an error above? Why are we getting zeros in the second version? Let's analyze the second version. Clearly, what is different here is the `(1/16**k)` in the front instead of division at the end.

In [3]:

```
1/16**270
```

Out[3]:

```
0.0
```

Above, is division of two **integers**. Python is being smart here and telling you that the **closest** floating point number to `1/16**270` is the floating point `0.0`. That's why we start seeing zeros.

Now, let's turn to the first version. There, we are evaluating a long expression and then dividing it by an integer. In essence, we take `float` and try to divide by an `int`. Recall that in Python 3.5, an `int` can be **arbitrarily large** (up to the amount of computer memory python is allowed to use). A `float`, however, only has a finite amount of space it can use, so there is a maximum.

In [4]:

```
float(16**270)
```

```
---------------------------------------------------------------------
---------
OverflowError                                Traceback (most recent c
all last)
<ipython-input-4-821a42bf104a> in <module>()
----> 1 float(16**270)

OverflowError: int too large to convert to float
```

So, when we call `1.0/16**270`, where we ask python to divide a `float` by an `int`, python **fist** tries to convert `16**270` to a `float` and fails!

In [5]:

```
1.0/16**270
```

```
---------------------------------------------------------------------
---------
OverflowError                                Traceback (most recent c
all last)
<ipython-input-5-578d4f96f2ba> in <module>()
----> 1 1.0/16**270

OverflowError: int too large to convert to float
```

Containers

Last time, we talked about the `list` type. We mostly looked at how to *build* **new** lists. We saw that we can create new lists by

- the `[]` constructor
- the `list()` function
- slicing of lists, i.e. `data[start, stop, step]`
- list comprehension

All of these methods returned **new** list objects. In fact, python gives each object a *unique identifier* which we can see by calling the `id()` function.

In [6]:

```
cold = ['england', 'finland']
warm = ['spain','greece']

visit = [cold, warm]
another_visit = [['england', 'finland'],['spain','greece']]
```

In [7]:

```
visit == another_visit
```

Out[7]:

```
True
```

In [8]:

```
id(visit)
```

Out[8]:

```
4431497160
```

In [9]:

```
id(another_visit)
```

Out[9]:

```
4431496648
```

In [10]:

```
# Compares is the variable names
# (or symbols) point to the same value in memory
visit is another_visit
```

Out[10]:

```
False
```

In [11]:

```
visit[0]
```

Out[11]:

```
['england', 'finland']
```

In [12]:

```
visit[0] is cold
```

Out[12]:

```
True
```

In [13]:

```
another_visit[0]
```

Out[13]:

```
['england', 'finland']
```

In [14]:

```
another_visit[0] is cold
```

Out[14]:

```
False
```

Warning : Python only guarantees that identifiers are *unique* for objects that are **still** in memory. Recall that an object is erased if no reference points to it. You should **always** use the `is` keyword over directly comparing identifiers.

In [15]:

```
a = 'a'
b = 'b'
id(a+b) == id(b+a)
```

Out[15]:

```
True
```

In [16]:

```
print(b+a)
print(a+b)
```

```
ba
ab
```

What is happening here is `a+b` is first created and its identifier is computed. Once this has happen, the memory where `a+b` was stored is cleared because nothing is referencing it. Next, `b+a` is created and it just so happens that it takes up the same slot in memory and ends up having the same identifier number as the already **destroyed** `a+b`.

In [17]:

```
a+b is b+a
```

Out[17]:

```
False
```

Above, the two objects are created and kept in memory until their locations are compared.

Lists are mutable

Since lists are a key tool for organizing data, constantly creating new lists is a waste of resources. Therefore, python allow you to **modify** the contents of a list **without** changing its identifier. Types that can be modified in this way are called **mutable**.

In [18]:

```
print("Before we modify, cold contains", cold, "and id", id(cold))
cold[1] = 'sweden'
print("After we modify, cold contains", cold, "and id", id(cold))
```

```
Before we modify, cold contains ['england', 'finland'] and id 4431
497608
After we modify, cold contains ['england', 'sweden'] and id 443149
7608
```

In [19]:

```
print("Before we modify, cold contains", cold, "and id", id(cold))
cold.append('norway')
print("After we modify, cold contains", cold, "and id", id(cold))
```

```
Before we modify, cold contains ['england', 'sweden'] and id 44314
97608
After we modify, cold contains ['england', 'sweden', 'norway'] and
id 4431497608
```

In [20]:

```
visit
```

Out[20]:

```
[['england', 'sweden', 'norway'], ['spain', 'greece']]
```

In [21]:

```
another_visit
```

Out[21]:

```
[['england', 'finland'], ['spain', 'greece']]
```

Here are some modifications you can do to a list `some_list`.

- `some_list.append(x)` will append the object `x` to the end of `some_list`.
- `some_list.insert(idx, x)` will insert the object `x` into `some_list` at index `idx`.
- `some_list.extend(other_list)` adds the objects in list `other_list` to the end of `some_list`.
- `some_list.remove(x)` deletes the **first** occurrence of an object **equivalent** to `x` from `some_list`.
- `some_list.pop(idx)` **removes and returns** the object at index `idx` in `some_list`. If called *without* an argument, this defaults to removing and returning the *last* element of `some_list`.
- `some_list.sort()` will sort the elements of `some_list` in ascending order.

In [22]:

```
other_warm = ['spain', 'greece']
print("Are warm and other_warm the same object?", warm is other_warm)
print("But are they equivalent?", warm == other_warm)
```

```
Are warm and other_warm the same object? False
But are they equivalent? True
```

In [23]:

```
print("The visit list before :", visit)
# Here we successfully remove the list equivalent to warm
visit.remove(other_warm)
print("The visit list after :", visit)
```

```
The visit list before : [['england', 'sweden', 'norway'], ['spain'
, 'greece']]
The visit list after : [['england', 'sweden', 'norway']]
```

In [24]:

```
print("The other_visit before", another_visit)
top = another_visit.pop(0)
print("The other_visit after", another_visit)
print("top is now", top)
```

```
The other_visit before [['england', 'finland'], ['spain', 'greece'
]]
The other_visit after [['spain', 'greece']]
top is now ['england', 'finland']
```

In [25]:

```
print("Warm before sorting", warm)
warm.sort()
print("Warm after sorting", warm)
```

```
Warm before sorting ['spain', 'greece']
Warm after sorting ['greece', 'spain']
```

In [26]:

```
['hello',7].sort() # Can't sort things that can't be compared
```

```
---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-26-b81d1e8f99f5> in <module>()
----> 1 ['hello',7].sort() # Can't sort things that can't be compa
red

TypeError: '<' not supported between instances of 'int' and 'str'
```

Remark 1

There are also version of some of these commands that will return **new** lists. For example

- `sorted(some_list)` will return a new sorted copy of `some_list` without touching the content of `some_list`
- you can *add* two lists using the + notation to create a new list instead of using `.extend()`
- if you need repeating list of some length, you can use the `*` notation with an `int` to create a repeating list

In [27]:

```
data = [67,47,57,37,10,20,311,232,23,1]
sorted_data = sorted(data)
# Note \n is the new line character
print("Original data is", data,
      "\n while sorted_data is", sorted_data)
```

```
Original data is [67, 47, 57, 37, 10, 20, 311, 232, 23, 1]
 while sorted_data is [1, 10, 20, 23, 37, 47, 57, 67, 232, 311]
```

In [28]:

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

Out[28]:

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

In [29]:

```
8*[1] # Awesome way to make a repeated list
```

Out[29]:

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

Remark 2 : copying

There might be times when you want to make a **copy** of an object. You should usually try to do this using the `.copy()` method (if the object has one). If `.copy()` does not exists, you can try the basic constructor, such as `list()`.

In [30]:

```
visit_copy = visit.copy()
print("Are visit and visit_copy the same object?", visit is visit_copy)
print("But are they equivalent?", visit == visit_copy)
```

```
Are visit and visit_copy the same object? False
But are they equivalent? True
```

In [31]:

```
# we can call the constructor of the type
# if no copy method is avaialble
visit_copy = list(visit)
print("Are visit and visit_copy the same object?", visit is visit_copy)
print("But are they equivalent?", visit == visit_copy)
```

```
Are visit and visit_copy the same object? False
But are they equivalent? True
```

In [32]:

```
# Note that the enteries refernce the same objects!
visit_copy[0] is visit[0]
```

Out[32]:

```
True
```

Remark 3 : tab completion

In IPython (or spyder, or any other decent editor), you can always use the <TAB> key to see which methods an object responds to. You can also tab complete variable names and method names.

In [ ]:

```
visit.<TAB>
```

Most objects have some "hidden" methods that you can discover by typing `._` followed by <TAB>.

In [ ]:

```
visit._<TAB>
```

Tuples

Tuples are an **immutable** version of lists. This means that a `tuple` **cannot be modified**. To create tuples you can use the `( )` notation. You can also convert list to tuples and vice-versa.

In [34]:

```
a = (2,3,4)
b = list(a)
c = tuple(b)
print(a,b,c)
print(type(a),type(b),type(c))
```

```
(2, 3, 4) [2, 3, 4] (2, 3, 4)
<class 'tuple'> <class 'list'> <class 'tuple'>
```

In [35]:

```
a[2] = 5
```

```
---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-35-dec41322dc24> in <module>()
----> 1 a[2] = 5

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

Warning : using the + and * notation on tuples works just like on lists. They are **not** vectors.

In [36]:

```
('hello',3,4)+(5,6,7)
```

Out[36]:

```
('hello', 3, 4, 5, 6, 7)
```

In [37]:

```
4*(1,3)
```

Out[37]:

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

Strings and characters

One can think of strings as **immutable** lists of characters that have many useful methods. A string can contain most forms of characters, including spaces and new lines. Python 3.5 allows your strings to contain pretty much any character from any language set (by default, strings are encoded using the UTF-8 character encoding). To create basic strings we use

- single quotes

```
desc = 'this is a string'
```

- double quotes

```
desc = "this is also a string, just like above"
```

- triple single or double quotes

```
desc = '''This string can span
several lines at once'''
desc = """You may choose whichever
style you like better"""
```

  Some style guides suggest that programmers use double quotes for textual output and single quotes for strings they will use or manipulate in their code.

In [38]:

```
vowels = 'aeiou'
desc = "These are all the vowels in the English alphabet :"
print(desc,vowels)
```

```
These are all the vowels in the English alphabet : aeiou
```

Strings are a container objects for characters, so they can be manipulated in many of the same ways.

In [39]:

```
vowels[::-2] # We can slice strings
```

Out[39]:

```
'uia'
```

In [40]:

```
len(vowels)
```

Out[40]:

```
5
```

In [41]:

```
vowels + 'é'
```

Out[41]:

```
'aeioué'
```

In [42]:

```
4*'repeat '
```

Out[42]:

```
'repeat repeat repeat repeat '
```

Strings have tons of useful methods. Here are a few. Remember that string are immutable, so all methods return new objects. Also, all search methods use == as the test.

- `some_string.count(sub_string)` counts how many times `sub_string` occurs in `some_string`.
- `some_string.find(sub_string)` returns the **non-negative** index of the first occurrence of `sub_string` in `some_string` and `-1` if not found.
- `some_string.rfind(sub_string)` is reverse find, same as find but starts from the end of `some_string`.
- `some_string.lower()` returns an all lowercase version of `some_string`.
- `some_string.replace(old, new)` returns a string with all occurrences `old` in `some_string` replaced with `new`.
- `some_string.rstrip()` returns a string with all trailing white space removed from `some_string`.
- `some_string.split(delim)` returns a list of strings cut along `delim`.
- `delim.join(list_of_strings)` return a string which is the concatenation of the strings in `list_of_strings` separated by `delim`.

In [43]:

```
word = 'esteemed'
# Note the single quote inside the double quote
print("The word", word, "has", word.count('e'), "e's in it")
```

```
The word esteemed has 4 e's in it
```

In [44]:

```
c = 'd'
idx = word.find(c)
print("Looking for character", c, "and found the character",
      word[idx], "at index", idx, "of string", word)
```

```
Looking for character d and found the character d at index 7 of st
ring esteemed
```

In [45]:

```
# Be sure to check the sign of you found index!
c = 'j'
idx = word.find(c)
print("Looking for character", c, "and found the character",
      word[idx], "at index", idx, "of string", word)
```

```
Looking for character j and found the character d at index -1 of s
tring esteemed
```

Clearly we did not find the right character. We can also search for substrings.

In [46]:

```
word.find('ee')
```

Out[46]:

```
3
```

In [47]:

```
# Stipping whitespace
(4*'repeat ').rstrip()
```

Out[47]:

```
'repeat repeat repeat repeat'
```

In [48]:

```
# Splitting stings can be super useful
'I hope we are not out of time yet'.split(' ')
```

Out[48]:

```
['I', 'hope', 'we', 'are', 'not', 'out', 'of', 'time', 'yet']
```

In [49]:

```
# Joining is the reverse of splitting. Also extremely useful
'|'.join(['a', 'list', 'of', 'words'])
```

Out[49]:

```
'a|list|of|words'
```

I recommend that you read about many of the other string manipulation methods available in python at docs.python.org/3.5/library/stdtypes.html

Some basic formatting

One of the most useful methods of the `str` type is the `.format()` method. It allows you to "print" or display the content of variables into a string. Let's start with an example.

In [50]:

```
desc = 'One {0} is about {1:.2%}. '.format('third', 1/3)
print(desc)
```

```
One third is about 33.33%.
```

Above, the brackets contain `{ index_or_key : format_spec }`. The `index_or_key` variable tells the string which argument of `format()` to place there. The `format_spec` tells the string how to format the variable. For example, `.2%` means we want a percentage with 2 places after the decimal. Note, currently we are using the US notion for numbers. We will learn to change this later by changing the `locale`.

In [51]:

```
# We can reuse the same variable multple times
# and display it in different formats
desc = 'The integer {0} can be formatted as a float {0:,.2f}'.format(132919321
)
print(desc)
```

```
The integer 132919321 can be formatted as a float 132,919,321.00
```

In [52]:

```
desc = '''We can also pad output with zeros {0:0>4}, {1:0<4},
 spaces in front |{0:>4}| or back |{1:<4}|,
 and with other characters |{0:j>5}| and pad all around |{1:j^10}|'''.format(6
,6**2)
print(desc)
```

```
We can also pad output with zeros 0006, 3600,
 spaces in front |   6| or back |36  |,
 and with other characters |jjjj6| and pad all around |jjjj36jjjj|
```

Formatting can get very complicated very quickly. For full details, read docs.python.org/3/library/string.html#formatspec. Here is a brief overview.

- Format spec can look like `[[fill]align][width][,][.precision][type]` **without** the grouping brackets `[]`
- `fill` can be any character
- `align` usually used as < (pad right), > (pad left), or ^ (center) is the `fill` character
- `width` is an integer specifying *minimum* field width
- `,` works only for printing comma separated floats and integers (US-style only)
- `precision` works differently depending on the type. For numerical types, it specifies the number of digits after the decimal point. For non-numerical types, it specifies the *maximum* field width.
- `type` can be many different options, common ones are
  - integers types are `d` (decimal), `b` (binary), `x` (hex), `n` (locale formatted)
  - floating point times are `f` (fixed point), `e` (scientific), `g` (general format), `%` (percent), `n` (locale formatted)
  - string types are `s` (string) and `c` (character)

In [53]:

```
'The integer {1:d} is {1:b} in binary'.format(232, 450)
```

Out[53]:

```
'The integer 450 is 111000010 in binary'
```

Control Flow

We can control the flow of a program using conditional statements and loops. Python uses **indentation** to indicate blocks of code withing a loop or any kind of statement. You will see how this works below.

if/elif/else

The `if` statement is relatively straight forward. When we encounter a true condition, we execute the assocaited block and then continue **after** the whole `if/elif/else` statement. Here, `elif` stnds for 'else if' and `else` is inidcates what to do if all other conditions are not satified. You can play with the `cold` variable below to see how things change.

In [54]:

```
cold = ['sweden', 'finland']
# if this is true do the next set of indented lines
if 'england' in cold :
    print("England is cold")
    print("First condition is true")
# else, if this second condition is true,
# do the next set of indented lines
elif 'finland' in cold :
    print("Finland is cold")
    print("Second condition is true and first is false")
# else, if this third condition is true,
# do the next set of indented lines
elif len(cold) == 2 :
    print("Only two cold places")
    print("Third condition is true, while first and second are both false")
else :
    print("Nothing is true")
```

```
Finland is cold
Second condition is true and first is false
```

You don't need to inlucde `elif` or even `else` statements in your if statements.

In [55]:

```
if 'blue' in 'the sky can be blue, red, or orange' :
    print("the sky can be blue")
print("I don't need an else or elif statement")
```

```
the sky can be blue
I don't need an else or elif statement
```

In [56]:

```
if 'brown' in 'the sky can be blue, red, or orange' :
    print("the sky can be brown")
else :
    print("the sky can't be brown")
```

```
the sky can't be brown
```

for/range

The `for` loop is useful when you need to iterate over a list or known set of indexes. **Do not modify what you are iterating over**.

In [57]:

```
for i in range(10) :
    print(i)
```

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

In [58]:

```
for c in 'some string' :
    print(c)
```

```
s
o
m
e

s
t
r
i
n
g
```

In [59]:

```
for p in 'comma,separated,string'.split(',') :
    print(p)
```

```
comma
separated
string
```

In [60]:

```
data = [1,2,3,4,5]
for x in data :
    print(x)
    data.remove(x) # DO NOT DO THIS!!!
```

```
1
3
5
```

If I used `.append(x)` above, I would be appending **forever**!

while/break/continue

The `while` loop is great if you do not know *how many* iterations you need to do. For example, we can play with the Collatz conjecture (also known as the Syracuse problem)

In [61]:

```
u = 10
seq = [u]
while u != 1 :
    # The loop block of code starts here
    if u % 2 == 0 and u > 0:
        u = u // 2
    else :
        u = 3*u + 1
    seq.append(u)
    print(seq)
    # The loop block of code ends here with the last indent of this level
# Now the while loop is completely over!
print('Starting with {0}, the Syracuse sequence terminates in {1} steps'.forma
t(seq[0], len(seq)-1))
print('The sequence is', seq)
```

```
[10, 5]
[10, 5, 16]
[10, 5, 16, 8]
[10, 5, 16, 8, 4]
[10, 5, 16, 8, 4, 2]
[10, 5, 16, 8, 4, 2, 1]
Starting with 10, the Syracuse sequence terminates in 6 steps
The sequence is [10, 5, 16, 8, 4, 2, 1]
```

There are also key words

- `break` tells the program to leave (or break from) the inner most `for` or `while` loop
- `continue` tells the program to skip the rest of the code block and start a new loop

In [62]:

```
n = 6
n_factorial = n # We will modify this variable in the loop
while True :
    if n != 1 :
        n = n-1
        n_factorial *= n # This is short hand for n_factorial = n_factorial *
n
    else :
        break # Leave the loop entirely
print(n_factorial)
```

```
720
```

In [63]:

```
for c in 'pythoneeeee' :
    print(c) # Just to see what the loop is doing, let's see what we are inter
ating over
    if c not in 'aeiou' :
        continue # Continue at the top of the loop with next value of c
    print('The first vowel is', c) # This only gets executed when the conditio
n in the if statement is false
    break # We found one vowel, we are done
```

```
p
y
t
h
o
The first vowel is o
```

Functions (briefly)

Functions are used to organize code into reusable units. For example, we can write a factorial function as

In [64]:

```
def factorial(n) :
    result = 1
    for x in range(n,1,-1) : # Note the range(n,1,-1) staring with n and stepp
ing back by 1
        result *= x # Again we use the short hand for result = result * x
    return result
```

In [65]:

```
factorial(5)
```

Out[65]:

```
120
```

In [66]:

```
print('{0}! is equal to {1}'.format(2,factorial(2)))
print('{0}! is equal to {1}'.format(10,factorial(10)))
print('{0}! is equal to {1}'.format(-1,factorial(-1)))
```

```
2! is equal to 2
10! is equal to 3628800
-1! is equal to 1
```

You can give a function multiple arguments

In [67]:

```
def discriminant(a,b,c) :
    return b**2 - 4*a*c
print(discriminant(3,4,7))
```

```
-68
```

We will get more into functions and arguments in the next lecture.

Modules (briefly)

All the code we have written so far has been the interpreter. For managing projects and more complex coding paths, we need to write the code in text files. These text files will be called either *scripts* or *modules*. For example, I have written a file called `useless_module.py` with one function inside called `useless`. I put this file in the current directory.

In [71]:

```
%ls
```

```
Lecture 2.ipynb    useless_module.py
```

In [72]:

```
%cat useless_module.py # Print the contents of the file to output
```

```
def useless() :
    print("I don't do anything useful yet!")
```

We can **import** the code in the file `useless_module.py` into our active session. Calling `import` acutally runs the code in your file.

In [73]:

```
import useless_module # import, that is run, the code in the file useless_module.py
useless_module.useless() # we can now call the uselss function we defined in useless_module.py
```

```
I don't do anything useful yet!
```

Python (and the Anaconda distribution) comes with a huge list of available modules and *packages* (which are collections of modules). Here are a few.

In [74]:

```
import math
import random
```

In [75]:

```
math.sin(30)
```

Out[75]:

```
-0.9880316240928618
```

In [76]:

```
perm = [1,2,3,4]
random.shuffle(perm) # Randomly shuffles the list
print(perm)
```

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

We will discover modules along the way. Your homework this week will be to write a module of your own.