Intoduction to Programming

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

Overview

The goal of this course is to provide an introduction to programming by focusing on practical tools, common practices and mathematical experiments. We will work almost entirely with the Python programing language. Python is a great general purpose language for scientific computing and fast prototyping. Additionally, we will learn how to use the git version control system. At the end of the course, students should be able to create and manage a small software project for scientific computing.

Along with basic programming skills and workflow, I hope we will be able to discuss some of the following topics: recursion, regular expressions, sorting algorithms, solving ODEs, trees, graph traversals and planarity, data manipulation and visualizations, machine learning and neural networks.

Prerequisites

No previous knowledge of computer programing is assumed. However, a good knowledge college level mathematics is required. Access to a personal computer for homework assignments and collaborative work is also necessary.

Software and Text

We will be working with Python 3.5 as part of the Anaconda distribution and the git version control system. You can find the links to download the necessary software below. You are free to use any other distribution of Python, but please try to resolve any compatibility issues yourself.

- Anaconda can be found at continuum.io/downloads.
- git version control software can be found at git-scm.com/downloads.
- other software we may use will be announced in class and on the website.

Our main reference for this course will be the SciPy Lecture Notes found at scipy-lectures.org. However, much of the material will be presented only in lecture.

You can also find many other resources for learning Python online. In particular, there is a French language text by Gérard Swinnen that students may find useful at inforef.be/swi/python.htm. Additionally, there is a great list of open access Python texts found at pythonbooks.revolunet.com.

Grading

The course will consist of weekly homework assignments, 2 (or 3) coding projects, and one final project. Homework and projects will be at first submitted via email and later using a git repository. You will collaborate on projects with other students in groups of two or three. For homework assignments, feel free to discuss the problems with other students, but the submitted work must be your own writing and code.

- 50 % - weekly homework assignments
- 30 % - 2 (or 3) coding projects
- 25 % - final project

Why Python?

- Works for both interactive and non-interactive workflows
- Large collection of ready-to-use tools and algorithms.
- Extensive online examples and learning tools.
- Easy to learn and straight forward to read. This is great for collaborative work.
- Open source. This is key in scientific computing. You don't want your accademic work to rely on bugs in closed sourced software.
- Extremely versatile. Everything from server-side clients to graphical programs.

Computers and Programming Languages

On a basic level, a computer is a machine that executes instructions. Our goal will be to learn how to write a form of the instruction in a high-level programming language. I want you to keep in mind some basic ideas about how a computer works as you write and debug your programs.

A computer

At the hardware level, a computer can do "primitive" operations but do them very quickly. The two main pieces of a modern computers are

- the central processing unit (cpu) - a device that can perform a finite number of operations on collections of (binary) numbers such as multiplication, addition, bit-wise and/or operations, etc. A cpu understands only *machine code*, which is a set of machine-level primitive instructions.
- memory - a place where data, instructions, and results of computations are stored.

If anyone is interested in the mathematical theory of computation and it's limitations, I suggest you read about Turing Machines.

A programming language

The purpose of a programming language is to abstract the simple operations performed by the cpu into a higher level language. The code you write is simply an organized collection of text files full of instructions. There are two ways to convert these files into machine code

- a compiler - a program that takes your code and generates a new programs, which is a list of machine code instructions, that can then run on the cpu.
- an interpreter - a program that interprets your text files a few lines at a time, converts them to machine code, and runs them.

Python is usually used as an interpreted language. For example, I can tell the Python interpreter to print out "Hello!".

In [1]:

```
print("Hello!")
```

```
Hello!
```

One advantage of Python that makes it easy to learn and use is that given a good interpreter (IPython in our case), you can ask it how certain instructions behave.

In [2]:

```
help(print)
```

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

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=Fa
lse)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sy
s.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline
.
    flush: whether to forcibly flush the stream.
```

You can also type in `print?` and then hit enter to get the same inforamtion. Another example of using print :

In [3]:

```
print(3.4)
```

```
3.4
```

Let's begin with the tools

For compatibility, we will us the Anaconda distribution of Python 3.5. Once you have installed the software package, you can open up the terminal to start using the IPython interpreter. Linux users should know where their terminal is, macOS users can find it in /Applications/Utilities/Terminal.app, and Windows users can use Windows Key + R, type in cmd.exe and hit run.

Once at the terminal, let's begin by starting IPython with just the simple command : ipython

In [4]:

```
2**1.5
```

Out[4]:

```
2.8284271247461903
```

In [5]:

```
5.5/4.2
```

Out[5]:

```
1.3095238095238095
```

In [6]:

```
(2017/500) % 4
```

Out[6]:

```
0.033999999999999981
```

You will find that you can quickly use python as a simple calculator with operations like : *, /, +, -, ** (exponentiate), % (remainder), and you can change the order of operations by using parentheses ().

You can also designate *variables* in python and assign values

In [7]:

```
a = 1.1
```

In [8]:

```
print(a)
```

```
1.1
```

Notice that I do not have to tell python that the variable "a" is a number, or even what kind of number. Python is strongly, dynamically typed language. This means that values that are stored in memory have an assigned type to them (integer, list, floating point number, etc) while the variables names themselves do not. How to we check the type of a variables value?

In [9]:

```
type(a)
```

Out[9]:

```
float
```

Numerical types

Python has four built-in numerical types : integers, booleans, floating point numbers, and floating point complex numbers.

As of Python 3, **integers** have no maximum or minimum values, so adding two large integers will not "overflow" the memory container to the integer value. This is not the case in Python 2.

In [10]:

```
type(3+3**3)
```

Out[10]:

```
int
```

Booleans simply take the values True and False. They are usually the return values of comparison operators. You can also use the operators *and*, *or*, and *not* with booleans.

In [11]:

```
type(True)
```

Out[11]:

```
bool
```

In [12]:

```
True or False
```

Out[12]:

```
True
```

In [13]:

```
3.5 > 3.5
```

Out[13]:

```
False
```

In [14]:

```
2 <= 2
```

Out[14]:

```
True
```

In [15]:

```
5 == 5
```

Out[15]:

```
True
```

In [16]:

```
not (3 > 2)
```

Out[16]:

```
False
```

Floating point numbers express a finite collection of rational numbers. A floating point number x is represented by three integers $s = \pm 1$, $c =$ significand, $q =$ exponent. Numerically

$$x = s \cdot c \cdot 2^q .$$

Due to size constraints, the numbers c and q have a finite number of possible integer values and, therefore, so does x. When you tell a computer to construct the number 6.6 in memory, it find the *closest* representable floating point number. Similarly, when a computer performs an operation on two floating point numbers, it finds the closest possible floating point representative to the "true" value. Here is something bizarre :

In [17]:

```
type(5.7)
```

Out[17]:

```
float
```

In [18]:

```
3.3 * 2.0 == 6.6
```

Out[18]:

```
True
```

In [19]:

```
2.2 * 3 == 6.6
```

Out[19]:

```
False
```

In [20]:

```
abs(6.6-(2.2*3.0))<1e-15
```

Out[20]:

```
True
```

Excercise Explain why you should *never* compare floats for equality!

Warning Division of integers in Python 3.5 using / will always return floats. If you want to know the integer quotient (i.e. $m = q \cdot n + r$) then use //. Other operations, such as exponentiation, may return a float or an integer depending on the input.

In [21]:

```
type(4/2)
```

Out[21]:

```
float
```

In [22]:

```
type(4//2)
```

Out[22]:

```
int
```

In [23]:

```
4//3
```

Out[23]:

```
1
```

In [24]:

```
17263 == (17263 // 3) * 3 + (17263 % 3)
```

Out[24]:

```
True
```

In [25]:

```
2**-4
```

Out[25]:

```
0.0625
```

In [26]:

```
2**4
```

Out[26]:

```
16
```

In [27]:

```
type(1.)
```

Out[27]:

```
float
```

Floating point complex numbers make use of *1j* to denote $\sqrt{-1}$.

In [28]:

```
a = 5.0 + 7.1*1j
```

In [29]:

```
b = 4.3 + 6.j
```

In [30]:

```
print(a)
```

```
(5+7.1j)
```

In [31]:

```
a**b
```

Out[31]:

```
(-6.710811464143892-34.19161079235317j)
```

In [32]:

```
a.conjugate()
```

Out[32]:

```
(5-7.1j)
```

Type casting

You can ask python to convert, to the best of it's abilities, between different types. This process is called type casting. There will be times that this process will fail and give an error.

```
In [33]:
```

```python
int(5.6)
```

```
Out[33]:

5
```

Checkout `help(int)` or `int?` for documentation

```
In [34]:
```

```python
float(4)
```

```
Out[34]:

4.0
```

```
In [35]:
```

```python
bool(0.1)
```

```
Out[35]:

True
```

```
In [36]:
```

```python
bool(a)
```

```
Out[36]:

True
```

```
In [37]:
```

```python
int('3')
```

```
Out[37]:

3
```

```
In [38]:
```

```python
type('3')
```

```
Out[38]:

str
```

In [39]:

```
str(2.36575)
```

Out[39]:

```
'2.36575'
```

In [40]:

```
float('4.5')
```

Out[40]:

```
4.5
```

In [41]:

```
float(5.0 + 7.1*1j)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-41-6c2388aff87f> in <module>()
----> 1 float(5.0 + 7.1*1j)

TypeError: can't convert complex to float
```

In [42]:

```
int("Hello")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-42-5cdea6865089> in <module>()
----> 1 int("Hello")

ValueError: invalid literal for int() with base 10: 'Hello'
```

Assignment operator

You may have noticed that

- == is used for *comparing* values.
- = is used to *assign* a value to a variable name.

There is also the *is* operator **which checks if two variable names point to the same value in memory.**

In [43]:

```
a = 4
b = a
a is b
```

Out[43]:

```
True
```

In [44]:

```
b = 4.0
float(a) == b
```

Out[44]:

```
True
```

In [45]:

```
a is b
```

Out[45]:

```
False
```

In [46]:

```
b = 4
a is b
```

Out[46]:

```
True
```

Remark Here, we see that the python interpreter does not copy the integer `4` to a new memory location, but cleverly uses the same `4` that it created ealier.

Containers

Some of the most important objects in the python programming language are containers such as lists, strings, tuples, dictionaries, and sets.

Lists

To construct a list, we use [ ] to surround the a comma separated list of contents.

In [47]:

```
L = ['red', 'green', 'blue', 'yellow', 5]
```

In [48]:

```
print(L)
```

```
['red', 'green', 'blue', 'yellow', 5]
```

We can now access individual objects in a list also using the [ ] notation

In [49]:

```
L[3]
```

Out[49]:

```
'yellow'
```

In [50]:

```
L[0]
```

Out[50]:

```
'red'
```

Indexing in python **starts with 0** !!! Even more importantly, python can accept **negative** indexes!

In [51]:

```
L[-2]
```

Out[51]:

```
'yellow'
```

The range of valid indexes is *-len(L)* to *len(L)-1*

In [52]:

```
len(L)
```

Out[52]:

```
5
```

In [53]:

```
L[len(L)]
```

```
---------------------------------------------------------------------------
----------
IndexError                                Traceback (most recent c
all last)
<ipython-input-53-6889782ef819> in <module>()
----> 1 L[len(L)]

IndexError: list index out of range
```

In [54]:

```
L[len(L)-1] is L[-1]
```

Out[54]:

```
True
```

List slicing

You can slice lists using the syntax *L[start:stop:stride]*. All slicing parameters are optional. Stop is not included in the output.

In [55]:

```
print(L)
```

```
['red', 'green', 'blue', 'yellow', 5]
```

In [56]:

```
D = [0,1,2,3,4,5,6,7,8,9]
D[::2]
```

Out[56]:

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

In [57]:

```
L[2::-2] # Go in reverse from index 2
```

Out[57]:

```
['blue', 'red']
```

In [58]:

```
L[::-1] # Reverse the whole list!
```

Out[58]:

```
[5, 'yellow', 'blue', 'green', 'red']
```

Building lists and list comprehension

There are many different ways to generate a lists. The simplest function is the *range* function. In Python 3, the function *range*(start, stop, stride) acually produces a special *iterable* type, or an object you can ask to produce a next *step*. Therefore, we have to cast the result to produce a list.

In [59]:

```
list(range(10))
```

Out[59]:

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

In [60]:

```
list(range(2,10))
```

Out[60]:

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

In [61]:

```
list(range(2,10,2))
```

Out[61]:

```
[2, 4, 6, 8]
```

One great way to build new lists from old is to use what is called list comprehension. From mathematical standpoint, this is very close to how mathematicians build sets.

In [62]:

```
cubes = [ x**3 for x in range(1,11)]
print(cubes)
```

```
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

In [63]:

```
sum(cubes)
```

Out[63]:

```
3025
```

Exercise Use the `sum()` to approximate π using the Bailey–Borwein–Plouffe formula

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

Use as many summands as you feel appropriate.