Lesson 1: Interacting with Python

This is the first lesson in our course *"Engineering Computations"*, a one-semester course for second-year university students. The course uses Python, and assumes no prior programming experience. Our first step will be to get you interacting with Python. But let's also learn some background.

1 What is Python?

Python is now 26 years old. Its creator, Guido van Rossum, named it after the British comedy "Monty Python's Flying Circus." His goals for the language were that it be an "an easy and intuitive language just as powerful as major competitors," producing computer code "that is as understandable as plain English."

It is a general-purpose language, which means that you can use it for anything: organizing data, scraping the web, creating websites, analyzing sounds, creating games, and of course *engineering computations*.

Python is an interpreted language. This means that you can write Python commands and the computer can execute those instructions directly. Other programming languages—like C, C++ and Fortran—require a previous *compilation* step: translating the commands into machine language. A neat ability of Python is to be used *interactively*. Fernando Perez famously created **IPython** as a side-project during his PhD. We're going to use IPython (the I stands for "interactive") in this lesson.

2 Why Python?

Because it's fun! With Python, the more you learn, the more you *want* to learn. You can find lots of resources online and, since Python is an open-source project, you'll also find a friendly community of people sharing their knowledge.

Python is known as a *high-productivity language*. As a programmer, you'll need less time to develop a solution with Python than with most languages. This is important to always bring up whenever someone complains that "Python is slow." Your time is more valuable than a machine's! (See the Recommended Readings section at the end.) And if we really need to speed up our program, we can re-write the slow parts in a compiled language afterwards. Because Python plays well with other languages :–)

The top technology companies use Python: Google, Facebook, Dropbox, Wikipedia, Yahoo!, YouTube... And this year, Python took the No. 1 spot in the interactive list of The 2017 Top

Programming Languages, by *IEEE Spectrum* (IEEE is the world's largest technical professional society).

Python is a versatile language, you can analyze data, build websites (e.g., Instagram, Mozilla, Pinterest), make art or music, etc. Because it is a versatile language, employers love Python: if you know Python they will want to hire you. **—Jessica McKellar, ex Director of the Python Software Foundation, in a 2014 tutorial.**

3 Let's get started

In this first lesson, we will be using IPython: a tool for working with Python interactively. If you have it installed in the computer you're using for this lesson, you enter the program by typing

```
ipython
```

on the command-line interface (the **Terminal** app on Mac OSX, and on Windows possibly the **PowerShell** or **git bash**). You will get a few lines of text about your IPython version and how to get help, and a blinking cursor next to the input line counter:

```
In[1]:
```

That input line is ready to receive any Python code to be executed interactively. The output of the code will be shown to you next to `Out[1]`, and so on for successive input/output lines.

Note: Our plan for this course is to work in a computer lab, where everyone will have a computer with everything installed ahead of time. For this reason, we won't discuss installation right now. Later on, when you're eager to work on your personal computer, we'll help you install everything you need. *It's all free!*

3.1 Your first program

In every programming class ever, your first program consists of printing a *"Hello"* message. In Python, you use the `print()` function, with your message inside quotation marks.

```
In [1]: print("Hello world!!")
```

```
Hello world!!
```

Easy peasy!! You just wrote your first program and you learned how to use the `print()` function. Yes, `print()` is a function: we pass the *argument* we want the function to act on, inside the parentheses. In the case above, we passed a *string*, which is a series of characters between quotation marks. Don't worry, we will come back to what strings are later on in this lesson.

Key concept: function A function is a compact collection of code that executes some action on its *arguments*. Every Python function has a *name*, used to call it, and takes its arguments inside round brackets. Some arguments may be optional (which means they have a default value defined inside the function), others are required. For example, the `print()` function has one required argument: the string of characters it should print out for you.

Python comes with many *built-in* functions, but you can also build your own. Chunking blocks of code into functions is one of the best strategies to deal with complex programs. It makes you more efficient, because you can reuse the code that you wrote into a function. Modularity and reuse are every programmer's friend.

3.2 Python as a calculator

Try any arithmetic operation in IPython. The symbols are what you would expect, except for the "raise-to-the-power-of" operator, which you obtain with two asterisks: `**`. Try all of these:

```
+   -   *   /   **   %   //
```

The `%` symbol is the *modulo* operator (divide and return remainder), and the double-slash is *floor division*.

```
In [2]: 2 + 2

Out[2]: 4

In [3]: 1.25 + 3.65

Out[3]: 4.9

In [4]: 5 - 3

Out[4]: 2

In [5]: 2 * 4

Out[5]: 8

In [6]: 7 / 2

Out[6]: 3.5

In [7]: 2**3

Out[7]: 8
```

Let's see an interesting case:

```
In [8]: 9**1/2

Out[8]: 4.5
```

Discuss with your neighbor: *What happened?* Isn't $9^{1/2} = 3$? (Raising to the power 1/2 is the same as taking the square root.) Did Python get this wrong?

Compare with this:

```
In [9]: 9**(1/2)
```

```
Out[9]: 3.0
```

Yes! The order of operations matters!

If you don't remember what we are talking about, review the Arithmetics/Order of operations. A frequent situation that exposes this is the following:

```
In [10]: 3 + 3 / 2
```

```
Out[10]: 4.5
```

```
In [11]: (3 + 3) / 2
```

```
Out[11]: 3.0
```

In the first case, we are adding 3 plus the number resulting of the operation 3/2. If we want the division to apply to the result of $3 + 3$, we need the parentheses.

Exercises: Use IPython (as a calculator) to solve the following two problems:

1. The volume of a sphere with radius r is $\frac{4}{3}\pi r^3$. What is the volume of a sphere with diameter 6.65 cm?

   For the value of π use 3.14159 (for now). Compare your answer with the solution up to 4 decimal numbers.

   Hint: 523.5983 is wrong and 615.9184 is also wrong.

2. Suppose the cover price of a book is $24.95, but bookstores get a 40% discount. Shipping costs $3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies? Compare your answer with the solution up to 2 decimal numbers.

```
In [ ]:
```

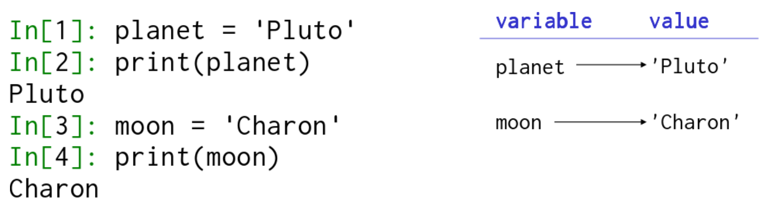
To reveal the answers, highlight the following line of text using the mouse:

Answer exercise 1: 153.9796 Answer exercise 2: 945.45

3.3 Variables and their type

Variables consist of two parts: a name and a value. When we want to give a variable its name and value, we use the equal sign: `name = value`. This is called an *assignment*. The name of the variable goes on the left and the value on the right.

The first thing to get used to is that the equal sign in an assignment has a different meaning than it has in Algebra! Think of it as an arrow pointing from `name` to `value`.



```
In[1]: planet = 'Pluto'
In[2]: print(planet)
Pluto
In[3]: moon = 'Charon'
In[4]: print(moon)
Charon
```

We have many possibilities for variable names: they can be made up of upper and lowercase letters, underscores and digits… although digits cannot go on the front of the name. For example, valid variable names are:

```
x
x1
X_2
name_3
NameLastname
```

Keep in mind, there are reserved words that you can't use; they are the special Python keywords.

OK. Let's assign some values to variables and do some operations with them:

```
In [12]: x = 3
```

```
In [13]: y = 4.5
```

Exercise: Print the values of the variables x and y.

```
In [ ]:
```

Let's do some arithmetic operations with our new variables:

```
In [14]: x + y
```

```
Out[14]: 7.5
```

```
In [15]: 2**x
```

```
Out[15]: 8
```

```
In [16]: y - 3
```

```
Out[16]: 1.5
```

And now, let's check the values of x and y. Are they still the same as they were when you assigned them?

```
In [17]: print(x)
```

```
3
```

```
In [18]: print(y)
```

```
4.5
```

3.4 String variables

In addition to name and value, Python variables have a *type*: the type of the value it refers to. For example, an integer value has type `int`, and a real number has type `float`. A string is a variable consisting of a sequence of characters marked by two quotes, and it has type `str`.

```
In [19]: z = 'this is a string'
In [20]: w = '1'
```

What if you try to "add" two strings?

```
In [21]: z + w
```

```
Out[21]: 'this is a string1'
```

The operation above is called *concatenation*: chaining two strings together into one. Insteresting, eh? But look at this:

```
In [22]: x + w
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-22-bc79f95cdaf2> in <module>()
    ----> 1 x + w


        TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Error! Why? Let's inspect what Python has to say and explore what is happening.

Python is a *dynamic language*, which means that you don't *need* to specify a type to invoke an existing object. The humorous nickname for this is "duck typing":

"If it looks like a duck, and quacks like a duck, then it's probably a duck." In other words, a variable has a type, but we don't need to specify it. It will just behave like it's supposed to when we operate with it (it'll quack and walk like nature intended it to).

But sometimes you need to make sure you know the type of a variable. Thankfully, Python offers a function to find out the type of a variable: `type()`.

```
In [23]: type(x)
```

```
Out[23]: int
```

```
In [24]: type(w)
```

```
Out[24]: str
```

```
In [25]: type(y)
```

```
Out[25]: float
```

3.5 More assignments

What if you want to assign to a new variable the result of an operation that involves other variables? Well, you totally can!

```
In [26]: sum_xy = x + y
         diff_xy = x - y
```

```
In [27]: print('The sum of x and y is:', sum_xy)
         print('The difference between x and y is:', diff_xy)
```

```
The sum of x and y is: 7.5
The difference between x and y is: -1.5
```

Notice what we did above: we used the `print()` function with a string message, followed by a variable, and Python printed a useful combination of the message and the variable value. This is a pro tip! You want to print for humans. Let's now check the type of the new variables we just created above:

```
In [28]: type(sum_xy)
```

```
Out[28]: float
```

```
In [29]: type(diff_xy)
```

```
Out[29]: float
```

Discuss with your neighbor: Can you summarize what we did above?

3.6 Special variables

Python has special variables that are built into the language. These are: `True`, `False`, `None` and `NotImplemented`. For now, we will look at just the first three of these.

Boolean variables are used to represent truth values, and they can take one of two possible values: `True` and `False`. *Logical expressions* return a boolean. Here is the simplest logical expression, using the keyword `not`:

```
not True
```

It returns... you guessed it... `False`.

The Python function `bool()` returns a truth value assigned to any argument. Any number other than zero has a truth value of `True`, as well as any nonempty string or list. The number zero and any empty string or list will have a truth value of `False`. Explore the `bool()` function with various arguments.

```
In [30]: bool(0)
```

```
Out[30]: False
```

```
In [31]: bool('Do we need oxygen?')
```

```
Out[31]: True
```

```
In [32]: bool('We do not need oxygen')
```

```
Out[32]: True
```

None is not Zero: `None` is a special variable indicating that no value was assigned or that a behavior is undefined. It is different than the value zero, an empty string, or some other nil value.

You can check that it is not zero by trying to add it to a number. Let's see what happens when we try that:

```
In [33]: a = None

         b = 3

In [34]: a + b
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-34-f96fb8f649b6> in <module>()
    ----> 1 a + b


        TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

3.7 Logical and comparison operators

The Python comparison operators are: `<`, `<=`, `>`, `>=`, `==`, `!=`. They compare two objects and return either `True` or `False`: smaller than, smaller or equal, greater than, greater or equal, equal, not equal. Try it!

```
In [35]: x = 3
         y = 5

In [36]: x > y

Out[36]: False
```

We can assign the truth value of a comparison operation to a new variable name:

```
In [37]: z = x > y

In [38]: z

Out[38]: False

In [39]: type(z)

Out[39]: bool
```

Logical operators are the following: `and`, `or`, and `not`. They work just like English (with the added bonus of being always consistent, not like English speakers!). A logical expression with `and` is `True` if both operands are true, and one with `or` is `True` when either operand is true. And the keyword `not` always negates the expression that follows.

Let's do some examples:

```
In [40]: a = 5
         b = 3
         c = 10
```

```
In [41]: a > b and b > c
```

```
Out[41]: False
```

Remember that the logical operator `and` is `True` only when both operands are `True`. In the case above the first operand is `True` but the second one is `False`.

If we try the `or` operation using the same operands we should get a `True`.

```
In [42]: a > b or b > c
```

```
Out[42]: True
```

And the negation of the second operand results in . . .

```
In [43]: not b > c
```

```
Out[43]: True
```

What if we negate the second operand in the `and` operation above?

Note: Be careful with the order of logical operations. The order of precedence in logic is:

1. Negation
2. And
3. Or

If you don't rememeber this, make sure to use parentheses to indicate the order you want.

Exercise: What is happening in the case below? Play around with logical operators and try some examples.

```
In [44]: a > b and not b > c
```

```
Out[44]: True
```

4 What we've learned

- Using the `print()` function. The concept of *function*.
- Using Python as a calculator.
- Concepts of variable, type, assignment.
- Special variables: `True`, `False`, `None`.
- Supported operations, logical operations.
- Reading error messages.

5 References

Throughout this course module, we will be drawing from the following references:

1. *Effective Computation in Physics: Field Guide to Research with Python* (2015). Anthony Scopatz & Kathryn D. Huff. O'Reilly Media, Inc.
2. *Python for Everybody: Exploring Data Using Python 3* (2016). Charles R. Severance. PDF available
3. *Think Python: How to Think Like a Computer Scientist* (2012). Allen Downey. Green Tea Press. PDF available

Recommended Readings

- "Yes, Python is Slow, and I Don't Care" by Nick Humrich, on Hackernoon. (Skip the part on microservices, which is a bit specialized, and continue after the photo of moving car lights.)
- "Why I Push for Python", by Prof. Lorena A. Barba (2014). This blog post got a bit of interest over at Hacker News.