Lecture 01: Python Basics

State Machines

Python interpreter works like a "State Machine". It has a current state, gets instructions, moves to the next state, etc.

> Initialise $\rightarrow$
>
> Prompt user for input $\rightarrow$
>
> Read what user types $\rightarrow$
>
> Interpret as expression $\rightarrow$
>
> Print results $\rightarrow$
>
> Prompt user for input...

State Machines are very powerful because

$$\text{State Machine} \ + \ \text{State Machine} \ = \ \text{State Machine}$$

Ie, if you feed the output of a State Machine into another State Machine, you get a new State Machine (assuming the second Machine knows what to do with the output of the first).

We're going to use and expand this idea a lot. Hopefully you'll start to appreciate how powerful of an idea it is. But first we are going to explore some simple exampels to get a feeling for the way Python works.

MAKE SURE YOU KNOW HOW TO DRIVE THE NOTEBOOK.

PUSH *"SHIFT ENTER"* TO EVALUATE A CELL

```
In [1]: **Markdown** *basics:*

        The markdown language is very simple to use.  Convert this cell from "Code" to
         "Markdown"

        You can enter in-line equations using dollar signs and LaTex math commands. Fo
        r example:  $e^{i \pi } + 1 = 0$

        You can enter centered equations with double-dollar signs. For example:

        $$ (x-1) \sum_{k=1}^{n} x^{-k} \ = \ 1 - x^{-n} $$

        You may not know it yet, but this formula is very important for decimal and bi
        nary representations of numbers.

        <br>
```

```
  File "<ipython-input-1-37cae04f7afa>", line 1
    **Markdown** *basics:*
     ^
SyntaxError: invalid syntax
```

NOTICE THE DROPDOWN BOX THAT SAYS "CODE", "MARKDOWN", ETC? MAKE SURE THIS IS SET TO THE RIGHT OPTION, OR NOTHING WILL WORK VERY WELL.

NOW FOR SOME EXAMPLES:

An 'integer'

```
In [ ]: 2
```

A 'float'

```
In [ ]: 2.1
```

You'll only see the output from the *last* thing in a cell.

```
In [ ]: 2
        2.1
```

Unless you explicitly ask for more:

```
In [ ]: print(2)
        print(2.1)
```

Note: The way you print things is one of the big difference between Python2 and Python3. We will use all Python3 in this course.

Here are some "magic words" to show the float with 15 digits

```
In [ ]: print('%.15f' %(2.1))
```

Magic words can show the float with 16 digits. Notice that 2.1 $\neq$ 21/10

```
In [ ]: print('%.16f' %(2.1))
```

Only numbers that are finitely represented in binary will avoid roundoff errors.

```
In [ ]: print('%.16f' %(1/512))
```

Try to say hello

```
In [ ]: hello
```

What happened?

Say 'hello' using a 'string'

```
In [ ]: 'hello'
```

Compositional systems

We can *combine* expressions. This seems very simple at first, but it is the basis for all the complexity we can create with computers.

EXAMPLES:

Combine integers

```
In [ ]: 1+2
```

Combine strings

```
In [ ]: 'Hello' + ' ' + 'world!'
```

Try to combine a string and an int

```
In [ ]: 'hello' + 2
```

```
In [ ]: 'hello' + '2'
```

Combine an int and a float

```
In [ ]: 3 + 2.1
```

Python will try to combine things in sensible ways if it can. This is *not* the most "rigorous" to do it. Some languages would convert the answer to a integer; becuase of significant figures reasons. But Python does it this way, and it's very handy.

What is the number more specifically?

```
In [ ]: print('%.16f' %( 3 + 2.1 ))
```

Sums and sentances (eg 'Hello world!') are examples of ***compositional systems***

```
In [ ]: 113*127 == 14351
```

```
In [ ]: 113*127 != 14351
```

int * int = int

```
In [ ]: (113*127) - 4 == 14347
```

(int * int) - int = int - int = int

If two things combine into some 'simpler thing', that also happens to be the same thing, then you can *replace* the more complicated thing with the simpler thing and trust everything still works.

The same is true of natural language.

Kanagroos *live in* **Australia.**

Pythons and kanagroos *live in* **Australia.**

Pythons and kanagroos *live and hunt for food in* **Australia.**

Pythons and kanagroos *live and hunt for food in* **Australia and New Guinea.**

NOUN + NOUN + *VERB* + *VERB* + **NOUN + NOUN** = **NOUN** + *VERB* + **NOUN**

Programming languages give you ways to make your own combinations. Suppose you find yourself doing something over and over again. You can give that thing a name and a definition, and then just use the name without worrying about how it's actually being done anymore.

For example

```
In [ ]: 1*1
```

```
In [ ]: 2*2
```

```
In [ ]: (2+1)*(2+1)
```

```
In [ ]: (11*17+4)*(11*17+4)
```

If we find ourselves repeating things again and again, it might be better to devise a system. This simiple principle forms the basis of all mathematics. Here is my favourite YouTube video about mathmatics. In my mind everything else follows from this:

https://www.youtube.com/watch?v=hgZwSRpfouQ (https://www.youtube.com/watch?v=hgZwSRpfouQ)

Think all you can about this and you'll be surpirsed how this really is ***everything*** in mathematics. Naming things to avoid confusion.

Function definitons

```
In [ ]: def square(n):
            return n*n
```

Notice the Python syntax for a function:

def function_name(arguements) colon

(4 spaces) intermidate computations

(4 spaces) return answer

The white space is essential

For simple functions you can do things in one line:

```
In [ ]: def square(n): return n*n
```

```
In [ ]: square(7.4)
```

```
In [ ]: square(11*17+4) == 36481
```

Now that the Python ***environment*** knows about square, we can use it just as easily as +, -, *, etc.

```
In [ ]: def sum_two_squares(n,m):
            return square(n) + square(m)
```

```
In [ ]: sum_two_squares(3,4)
```

```
In [ ]: sum_two_squares(3,4) == square(5)
```

What if we define another function into the enviroment?

```
In [ ]: def sum_two_cubes(n,m):
            return cube(n) + cube(m)
```

No complaining. But what happens if we try to run it?

```
In [ ]: sum_two_cubes(3,4)
```

READ THE ERROR MESSAGES CAREFULLY! They tend to be really useful in Python. It tells you the first thing that the interpreter didn't understand. This is usually enough to tell you what to fix.

In above example the `cube` function wasn't defined, even though `sum_two_cubes` is defined.

Hierarchical programming

The above is an example of building **hierarchical control structures**. For example, combining two or more functions into a new function.

We can also build **hierarchical data structures**.

Lists are a very important data structure in Python. The list is probably the simplest example that isn't a primitive data type, i.e., float, int, string.

For example

A **list** of integers

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

We put square brackets `[ ]` around *lists*.

A list can be *heterogeneous*

```
In [ ]: [2,'even prime',0.618033988749895,'special']
```

List of lists come up all the time

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

```
In [ ]: print( len( [[1,2,3], [4,5,6], [7,8,9]] ))
```

We can compose list, but maybe not in the way you migth first think.

```
In [ ]: [1,2,3] + [4,5,6]
```

This can be useful in ways you might not frist expect. Think of a shopping list.

```
In [ ]: [1,2,3] + [4,5,6,7,8,9,10,11]
```

With **functions**, we gave names to **control structures**.

With **variables**, we can give names to **data structures**.

For example:

An *integer*

```
In [ ]: n = 1
```

A *float*

```
In [ ]: x = 0.618033988749895 + n
```

A simple function of a variable

```
In [ ]: 1/x
```

Notice anything special?

We can give names to lists

```
In [ ]: L = [[1,2,3], [4,5,6], [7,8,9]]
```

Python has was of making lists talk. The brackets after a variable name is called `get_item`. It gets different thing depending on the context. In the case of lists:

```
In [ ]: L[1]
```

```
In [ ]: L[2]
```

```
In [ ]: L[3]
```

What happened?

```
In [ ]: L[0]
```

```
In [ ]: L[0][0]
```

```
In [ ]: L[0][1]
```

```
In [ ]: L[0][2]
```

```
In [ ]: L[0][-1]
```

```
In [ ]: L[-1]
```

In [ ]:
```
L[-2]
```

In [ ]:
```
L[-3] == L[0]
```

Positive indices start counting from 0 and go forward. Negative indices start counting from -1 and go backward. Getting the last element in a list (without knowing the length of the list) is very useful.

It's turtles all the way down...

In [ ]:
```
Q = ['turtle',['turtle',['turtle',['turtle','turtle']]]]
print(Q)
print(Q[0])
print(Q[1])
print(Q[1][0])
print(Q[1][1])
print(Q[1][1][0])
print(Q[1][1][1])
print(Q[1][1][1][0])
print(Q[1][1][1][1])
```

Q: What is the Python interpreter actually doing with all the variables and functions?

A: It's using a giant data structure called a ***binding enviroment***, aka, just an *enviroment*. This data structure is very simple, and forms the basis for a huge amount of things that computers do. UNIX operating systems basically work the same way.

Python uses a big table of names and values. The names are *binded* to the value. Hening binding enviroment.

$$ \begin{aligned} \mathrm{n} \quad & \to \quad 1 \ \mathrm{x} \quad & \to \quad 1.618033988749895 \ \mathrm{L} \quad & \to \quad [[1,2,3],\ [4,5,6],\ [7,8,9]] \

- \quad & \to \quad \mathrm{a\ function\ that\ knows\ how\ to\ make\ plus}\
- \quad & \to \quad \mathrm{a\ function\ that\ knows\ how\ to\ make\ times}\ \mathrm{square} \quad & \to \quad \mathrm{everything\ inside\ our\ function\ called \ square}\ \mathrm{sum_two_squares} \quad & \to \quad \mathrm{everything\ inside\ our\ function\ called \ sum_two_squares}\ & \ldots \ & \ldots \ & \ldots \ \mathrm{lots \ and \ lots \ more} \ & \ldots \ & \ldots \ \end{aligned} $$

It works a lot like the bouncer and guest list at a party. The interpreter (bouncer) looks at the environment (guest list). If it finds a name on the list, it does whatever that name is binded to and prints the result. If it doesn't find the name on the list, it complains. And that all!

The reason it's so powerful is that the right-hand side of a binding can be ***an entirely new enviroment***, and that new enviroment can contain further enviroments. And those daugheter enviroments can point back at the original enviroment.

For example,

$$\text{``everything inside our function called square''}$$

$$\text{needs}$$

$$* \quad (\text{from our original enviroment})$$

Everything works long as the interpreter can get to something it knows at the bottom.

Well, you might say: "Well, that seems really simple! Just a list of names and values. I thought computer science was hard, like math. Why don't you do it some fancy way?"

Well, the best minds have thought about this for decades. And the answer to the question is "*Because the fancy way doensn't exist!*" This is really all there is.

Of course, this is somewhat of a fib. Languages like C/C++ and FORTRAN don't work with this kind of model. They work by putting things in preassigned locations "*addresses*". Whenever you ask for something in C you are asking for the thing that is located at a given spot in memory. It's a minor difference, but make programming much more annoying.

Data hiding

Enviroments allow effecient **data hiding**. Recall that n=1 is something in the current enviroment. We can still make a function that takes an arguement called n, and nothing should go wrong.

Remember, n, is a both a variable in the current name space, and an arguement in a function. Check and see:

```
In [ ]: n
```

```
In [ ]: def f(n): return 2*n
```

```
In [ ]: f(2)
```

Let's add a few more variables to our enviroment.

```
In [ ]: a = 7.2e6
        b = 1.2e-3
        c = 3.1 + 0.2*1j
        print(a)
        print(b)
        print(c)
```

We can take the real and imaginary parts of c in the following way

```
In [ ]: c.real
```

```
In [ ]: c.imag
```

We'll learn more about the `.something` operation that appears after variable.

Notice that I slipped in some scientific notation. Ie,

$$\mathrm{a\,e\,n} \;\equiv\; \mathrm{a} \times 10^{\mathrm{n}}$$

a can be an `int` or `float`, n must be an `int`.

I also slipped in some complex numbers. The symbol

$$1\mathrm{j} \;\equiv\; \sqrt{-1}$$

is Python's way of saying i.

We can add and multiply all these different numbers with eachother. Python will cast the result to the most conservative thing it can.

```
int + int = int * int = int

float + float = float * float = float

complex + complex = complex * complex = complex

int + float = int * float = float

int + complex = int * complex = complex

float + complex = float * complex = complex
```

You can think of this as a kind of "right of way" for data types.

For example:

In [ ]:
```
print(1 + b)
print(a + b)
print(1 + c)
print(-3 + b)
print(c*c)
```

Even though n, a, b, and c are all in our current enviroment (aka *name space*), we can make a function that uses some of these values but not others.

For example:

Here is a function that doesn't conflict with n, a, b, or c.

In [ ]:
```
poly?
```

In [ ]:
```
def poly(n):
    """You can put a 'doc string' in a function in this way.
    You can put info about the function and what it does.
    In this case it computes a cubic polynomial; with the following...

    Parameters
    ----------
    n : int, float, or complex """

    a, b, c = 6, 2, 3

    # You can put comments in a function with the #hastag
    # In this function you might notice I definied three parameters, a, b, c.
    # You can put them all on one line in the way I did it.
    # Or you could put them all on multiple lines.

    # Defining a, b, c in this function won't hurt the other a, b, c,
    # becuase this is a different enviroment.

    #The following code will redefine the internal parameters. But only intern
ally.

    print("Original values: (a,b,c) = (%f, %f, %f)"  %(a,b,c))

    a = 1/a
    b = 1/b
    c = 1/c

    print(" Working values: (a,b,c) = (%f, %f, %f)"  %(a,b,c))

    # While, it's wastefull in this case. It let's you see different ways of d
oing the same thing.
    # We normally would have defined things simply as
    # a, b, c = 1/6, 1/2, 1/3.

    return a*n + b*square(n) + c*n*square(n)
```

You can't 'see' the internal comments of a function. But you can read the doc string in the following way

In [ ]:
```
?poly
```

In [ ]:
```
poly(9)
```

In [ ]:
```
print("Your vaules: (a,b,c) = (%f, %f)"  %(a,b))
```

In [ ]:
```
poly(9) == 9/6 + 9*9/2 + 9*9*9/3 == 285.
```

The function 'poly' doesn't know about our enviroments n, a, b, c. Every enviroment will use the first names it finds for something.

Here is a function that knows more about our variables.

```
In [ ]: print(a)
```

```
In [ ]: def multi_enviroment_poly(n):
            # These are locally defined variables.
            # The function will need to know 'a' from somewhere else.
            b, c = 1/2, 1/3
            return a*n + b*square(n) + c*n*square(n)
```

```
In [ ]: multi_enviroment_poly(9)
```

You can get a sense that it's using a = 7.2e6 becuase the output is much larger.

```
In [ ]: multi_enviroment_poly(9) == a*9 + 9*9/2 + 9*9*9/3
```

When the function 'multi_enviroment_poly' didn't find the value 'a' in it's own enviroment, it went looking for it one enviroment up, and it found our largeer value.

It's can be unsafe for function to rely on variables defined in this way. It would be much better in most cases to *pass* the 'a' variable to the function.

```
In [ ]: def pass_me_an_a_poly(a,n):
            # These are locally defined variables.
            # The function will get 'a' when you call it.
            b, c = 1/2, 1/3
            return a*n + b*square(n) + c*n*square(n)
```

Because 'a' is in the enviroment it will use the current value anytime it shows up

```
In [ ]: pass_me_an_a_poly(a,9)
```

But now we can use a different values if we need:

```
In [ ]: pass_me_an_a_poly(10,9) == 10*9 + 9*9/2 + 9*9*9/3
```

In the 15th century, they didn't really have surnames the way we do today. Leonardo da Vinci didn't think about his last name as "da Vinci". And to what ever degree he did think of this as his name, no one ever called him just 'da Vinci', the way people sometimes do today.

Vinci was where he was from. It was his *enviroment*. His name was Leonardo and that was that. But if anyone became confused, he could include where he was from to make it clear.

There is also another (older) famous Leonardo: Leonardo de Pisa. We know him as Fibonacci. But this was a name made up in the 19th centrury. I mention this because we are going to talk a lot about Fibinacci numbers and sequences a little later in the course.

Lke people, there are only so many good names to go around. It's better to keep reuse names and keep track of everything by their enviroment. Of course, families are just different kinds of enviroments. Just not ones that are tied 100% to a geographic location.