



TEXAS ADVANCED COMPUTING CENTER

[WWW.TACC.UTEXAS.EDU](http://WWW.TACC.UTEXAS.EDU)



TEXAS

The University of Texas at Austin

# Python 101

S. Charlie Dey, Director of Training and Professional Development

# *Agenda*

Welcome to Python

Variables and Data Types

Arithmetic Operations

Output and File/IO

Control Flow

Lists

User Defined Functions

Anonymous Functions

**<https://jupyter.designsafe-ci.org/>**

# ***What are Jupyter Notebooks?***

A web-based, interactive computing tool for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results.

# *How do Jupyter Notebooks Work?*

An open notebook has exactly one interactive session connected to a kernel which will execute code sent by the user and communicate back results. This kernel remains active if the web browser window is closed, and reopening the same notebook from the dashboard will reconnect the web application to the same kernel.

What's this mean?

Notebooks are an interface to kernel, the kernel executes your code and outputs back to you through the notebook. The kernel is essentially our programming language we wish to interface with.

# *Jupyter Notebooks, Structure*

- Code Cells

Code cells allow you to enter and run code

Run a code cell using Shift-Enter

- Markdown Cells

Text can be added to Jupyter Notebooks using Markdown cells. Markdown is a popular markup language that is a superset of HTML.

# *Jupyter Notebooks, Structure*

- Markdown Cells

You can add headings:

- # Heading 1
- # Heading 2
- ## Heading 2.1
- ## Heading 2.2

You can add lists

- 1. First ordered list item
- 2. Another item
  - · \* Unordered sub-list.
- 1. Actual numbers don't matter, just that it's a number
  - · 1. Ordered sub-list
- 4. And another item.

# *Jupyter Notebooks, Structure*

- Markdown Cells

pure HTML

```
<dl>
```

```
<dt>Definition list</dt>
```

```
<dd>Is something people use sometimes.</dd>
```

```
<dt>Markdown in HTML</dt>
```

```
<dd>Does *not* work **very** well. Use HTML <em>tags</em>.</dd>
```

```
</dl>
```

And even, Latex!

$$e^{i\pi} + 1 = 0$$



# *Jupyter Notebooks, Workflow*

Typically, you will work on a computational problem in pieces, organizing related ideas into cells and moving forward once previous parts work correctly. This is much more convenient for interactive exploration than breaking up a computation into scripts that must be executed together, as was previously necessary, especially if parts of them take a long time to run.

# *Jupyter Notebooks, Workflow*

Let a traditional paper lab notebook be your guide:

Each notebook keeps a historical (and dated) record of the analysis as it's being explored.

The notebook is not meant to be anything other than a place for experimentation and development.

Notebooks can be split when they get too long.

Notebooks can be split by topic, if it makes sense.

# *Jupyter Notebooks, Shortcuts*

- **Shift-Enter**: run cell
  - Execute the current cell, show output (if any), and jump to the next cell below. If **Shift-Enter** is invoked on the last cell, a new code cell will also be created. Note that in the notebook, typing **Enter** on its own *never* forces execution, but rather just inserts a new line in the current cell. **Shift-Enter** is equivalent to clicking the **Cell | Run** menu item.

# *Jupyter Notebooks, Shortcuts*

- **Ctrl-Enter**: run cell in-place
  - Execute the current cell as if it were in “terminal mode”, where any output is shown, but the cursor *remains* in the current cell. The cell’s entire contents are selected after execution, so you can just start typing and only the new input will be in the cell. This is convenient for doing quick experiments in place, or for querying things like filesystem content, without needing to create additional cells that you may not want to be saved in the notebook.

# *Jupyter Notebooks, Shortcuts*

- **Alt-Enter**: run cell, insert below
  - Executes the current cell, shows the output, and inserts a *new* cell between the current cell and the cell below (if one exists). (shortcut for the sequence **Shift-Enter**, **Ctrl-m a**. (**Ctrl-m a** adds a new cell above the current one.))
- **Esc** and **Enter**: Command mode and edit mode
  - In command mode, you can easily navigate around the notebook using keyboard shortcuts. In edit mode, you can edit text in cells.

# ***Introduction to Python***

**Hello World!**

**Data types**

**Variables**

**Arithmetic operations**

**Relational operations**

**Input/Output**

**Control Flow**

Do not forget:

Indentation matters!

# *Python*

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

Let's type that line of code into a Code Cell, and hit Shift-Enter:

```
Hello World!
```

# *Python*

```
print(5)
```

```
print(1+1)
```

Let's add the above into another Code Cell, and hit Shift-Enter

5

2



# *Python - Variables*

You will need to store data into variables

You can use those variables later on

You can perform operations with those variables

Variables are declared with a **name**, followed by '=' and a **value**

An integer, string,...

When declaring a variable, **capitalization** is important:

'A' <> 'a'

# *Python - Variables*

in a code cell:

```
five = 5
one = 1
twodot = 2.0
print (five)
print (one + one)
message = "This is a string"
print (message)
```

Notice: We're not "typing" our variables, we're just setting them and allowing Python to type them for us.

# ***Python - Data Types***

in a code cell:

```
integer_variable = 100  
floating_point_variable = 100.0  
string_variable = "Name"
```

Notice: We're not "typing" our variables, we're just setting them and allowing Python to type them for us.

# ***Python - Data Types***

Variables have a type

You can check the type of a variable by using the `type()` function:

```
print (type(integer_variable))
```

It is also possible to change the type of some basic types:

**`str(int/float)`: converts an integer/float to a string**

**`int(str)`: converts a string to an integer**

**`float(str)`: converts a string to a float**

Be careful: you can only convert data that actually makes sense to be transformed

# ***Python - Arithmetic Operations***

+	Addition	$1 + 1 = 2$
-	Subtraction	$5 - 3 = 2$
/	Division	$4 / 2 = 2$
%	Modulo	$5 \% 2 = 1$
*	Multiplication	$5 * 2 = 10$
//	Floor division	$5 // 2 = 2$
**	To the power of	$2 ** 3 = 8$

# ***Python - Arithmetic Operations***

Some experiments:

```
print (5/2)
print (5.0/2)
print ("hello" + "world")
print ("some" + 1)
print ("number" * 5)
print (3+5*2)
```

# *Python - Arithmetic Operations*

Some more experiments:

```
number1 = 5.0/2
```

```
number2 = 5/2
```

what **type()** are they?

```
type(number1)
```

```
type(number2)
```

now, convert number2 to an integer:

```
int(number2)
```

# ***Python – Reading from the Keyboard***

Let put the following into a new Code Cell:

```
numIn = input("Please enter a number: ")
```

Let's run this cell!



# ***Python – Reading from the Keyboard***

Let put the following into a new Code Cell:

```
stringIn = input("Please enter a string: ")
```

Let's run this cell!

put the word **Hello** as your input.

What happened?

# ***Python – Making the output prettier***

Let put the following into a new Code Cell:

```
print ("The number that you wrote was : ", numIn)
print ("The number that you wrote was : %d" % numIn)
```

```
print ("the string you entered was: ", stringIn)
print ("the string you entered was: %s" % stringIn)
```

Want to make it prettier?

    \n   for a new line

    \t   to insert a tab

```
print (" your string: %s\n your number: %d", %(numIn, stringIn))
```

for floating points, use %f

# ***Python – Writing to a File***

Let put the following into a new Code Cell:

```
my_file = open("output_file.txt", 'w')
var1 = "This is a string\n"
my_file.write(var1)
var2 = 10
my_file.write("\n")
my_file.write(str(var2))
var3 = 20.0
my_file.write("\n")
my_file.write(str(var3))
my_file.close()
```

# ***Python – Reading from a File***

When opening a file, you need to decide “how” you want to open it:

Just read?

Are you going to write to the file?

If the file already exists, what do you want to do with it?

r     read only (default)

w     write mode: file will be overwritten if it already exists

a     append mode: data will be appended to the existing file

# ***Python – Reading from a File***

Let's read from the file we created in the previous cell.

```
my_file = open("output_file.txt",'r')
content = my_file.read()
print(content)
my_file.close()
```

# ***Python – Reading from a File***

Let's read it line by line

```
my_file = open("output_file.txt",'r')
var1 = my_file.readline()
var2 = my_file.readline()
var3 = my_file.readline()
var4 = my_file.readline()
print("String: ", var1)
print("Blank: ", var2)
print("Integer: ", var3)
print("Float: ", var4)
my_file.close()
```

# ***Python – Reading from a File***

Tweak it a bit to make the code easier to read... introducing 'with'!

‘with’ will very nicely close your file for you

(Note the indentation!!)

```
with open("output_file.txt",'r') as f:
    var5 = f.readline()
    var6 = f.readline()
    var7 = f.readline()
    var 8 = f.readline()
    print("String: ", var5)
    print("Blank: ", var6)
    print("Integer: ", var7)
    print("Float: ", var8)
```

# ***Python – Control Flow***

So far we have been writing instruction after instruction where every instruction is executed

What happens if we want to have instructions that are only executed if a given condition is true?



# *Python - if/else/elif*

Let's look at some example of booleans.

type the following into a code cell

```
a = 2
```

```
b = 5
```

```
print (a>b)
```

```
print (a<b)
```

```
print (a == b)
```

```
print (a != b)
```

```
print (b>a or a==b)
```

```
print (b<a and a==b)
```

# ***Python – if/else/elif***

The if/else construction allows you to define conditions in your program

(Don't forget your indentation!!)

```
if conditionA:
    statementA
elif conditionB:
    statementB
else:
    statementD
this line will always be executed (after the if/else)
```

# ***Python – if/else/elif***

The if/else construction allows you to define conditions in your program

(Indentation is IMPORTANT!)

```
if conditionA:
    statementA
elif conditionB:
    statementB
else:
    statementD
this line will always be executed (after the if/else)
```

conditions are a datatype known as booleans, they can only be true or false

# *Python - if/else/elif*

A simple example

```
simple_input = input("Please enter a number: ")
if (int(simple_input)>10):
    print ("You entered a number greater than 10")
else:
    print ("you entered a number less than 10")
```

# *Python - if/else/elif*

You can also nest if statements together:

```
if (condition1):  
    statement1  
    if (condition2):  
        statement2  
    else:  
        if (condition3):  
            statement3 # when is this statement executed?  
else: # which 'if' does this 'else' belong to?  
    statement4 # when is this statement executed?
```

# Exercise:

enter a number from the keyboard into a variable.

using type casting and if statements, determine if the number is even or odd

# ***Python – For Loops***

When we need to iterate, execute the same set of instructions over and over again... we need to loop! (and introducing range())

(Indentation is IMPORTANT!)

```
for x in range(0, 3):  
    print ("Let's go %d" % x)
```

# ***Python – For Loops, nested loops***

When we need to iterate, execute the same set of instructions over and over again... we need to loop! and introducing range()

```
for x in range(0, 3):  
    for y in range(0,5):  
        print ("Let's go %d %d" % (x,y))
```



# Exercise:

using nested for-loops and nested if statements, write a program that loops from 3 to 1000 and print out the number if it is a prime number.

# Exercise:

using a for loop, find the triples that satisfies:

$$a*a + b*b = c*c$$

where

$$0 < a < 100$$

$$0 < b < 100$$

# ***Python - While Loops***

Sometimes we need to loop while a condition is true...

(remember to indent!)

```
i = 0                # Initialization
while (i < 10):      # Condition
    print (i)        # do_something
    i = i + 1        # Why do we need this?
```

# Exercise:

using a while loop, find the prime numbers less than 1000

# *Python - lists*

A list is a sequence, where each element is assigned a position (index)

First position is 0. You can access each position using []

Elements in the list can be of different type

```
mylist1 = ["first item", "second item"]
mylist2 = [1, 2, 3, 4]
mylist3 = ["first", "second", 3]
print(mylist1[0], mylist1[1])
print(mylist2[0])
print(mylist3)
print(mylist3[0], mylist3[1], mylist3[2])
print(mylist2[0] + mylist3[2])
```

# *Python - lists*

It's possible to use slicing:

```
print(mylist3[0:3])  
print(mylist3)
```

To change the value of an element in a list, simply assign it a new value:

```
mylist3[0] = 10  
print(mylist3)
```

# *Python - lists*

There's a function that returns the number of elements in a list

```
len(mylist2)
```

Check if a value exists in a list:

```
1 in mylist2
```

Delete an element

```
len(mylist2)  
del mylist2[0]  
print(mylist2)
```

Iterate over the elements of a list:

```
for x in mylist2:  
    print(x)
```

# *Python - lists*

There are more functions

```
max(mylist), min(mylist)
```

It's possible to add new elements to a list:

```
my_list.append(new_item)
```

We know how to find if an element exists, there's a way to return the position of that element:

```
my_list.index(item)
```

Or how many times a given item appears in the list:

```
my_list.count(item)
```



# Exercise:

create a 3 lists:

one list, x, holding numbers going from 0 to 100

one list, y1, holding  $x*x$

one list, y2, holding  $x*x*x$

write these out to a file with the format:

x, y1, y2

# ***Python – user defined functions***

User-defined functions are reusable code blocks; they only need to be written once, then they can be used multiple times. They can even be used in other applications, too.

These functions are very useful, from writing common utilities to specific business logic. These functions can also be modified per requirement. The code is usually well organized, easy to maintain, and developer-friendly.

As user-defined functions can be written independently, the tasks of a project can be distributed for rapid application development.

A well-defined and thoughtfully written user-defined function can ease the application development process.

# ***Python – user defined functions***

Step 1: Declare the function with the keyword `def` followed by the function name.

Step 2: Write the arguments inside the opening and closing parentheses of the function, and end the declaration with a colon.

Step 3: Add the program statements to be executed.

Step 4: End the function with/without return statement.

# *Python – user defined functions*

```
def userDefFunction (arg1, arg2, arg3
...):
    program statement1
    program statement3
    program statement3
    ....
    return;
```

# Exercise:

write a user defined function that accepts an integer as an argument then prints out that many number of prime numbers

# Exercise:

write a user defined function that accepts an integer as a parameter then returns the next prime number.

# Exercise:

A prime gap is the difference between two successive prime numbers. The  $n$ -th prime gap, denoted  $g_n$  or  $g(p_n)$  is the difference between the  $(n + 1)$ -th and the  $n$ -th prime numbers

Write a program that uses your prime number generator functions and print out the first set of prime numbers where the prime gap is greater than 13

# ***Python – Anonymous Functions***

type the following into a cell:

```
x = lambda a: a * 10
```

```
print (x(10))
```



# ***Python – Anonymous Functions***

try the following definition:

```
def myfunc(x):  
    return lambda a: a*x
```

```
y = myfunc(10)  
print (y(5))  
z = myfunc(100)  
print (z(5))
```

Questions? Comments?