1. BASICS OF PYTHON

JHU Physics & Astronomy Python Workshop 2015

Lecturer: Mubdi Rahman

HOW IS THIS WORKSHOP GOING TO WORK?

We will be going over all the basics you need to get started and get productive in Python! **Please code along with us as we go!**

There are likely multiple ways of doing many things in Python, but we're going to show you one way. It may not be the best for your particular purposes, but we will try to be self consistent.

We will constantly refer you back to the documentation. The packages we have here have **far** more functionality than the scope of this workshop. If there's something that you want to do, Python likely has a package or function that can do it (or at least make your life easier).

HOW IS THIS WORKSHOP GOING TO WORK?

We will be going over all the basics you need to get started and get productive in Python! **Please code along with us as we go!**

There are likely multiple ways of doing
we're going to show you one way. It ma
particular purposes, but we will try to be

PRO TIP:

These boxes will have useful tidbits about python conventions or hints on how to make your coding life easier!

We will constantly refer you back to the
we have here have **far** more functionality than the scope of this workshop. If there's something that you want to do, Python likely has a package or function that can do it (or at least make your life easier).

WHY PYTHON?

Open Source/Free: No need to worry about licences

Cross-platform: Can be used with Windows/Macs OS/Linux

Full-featured Packages: If there's something you want to do, there's probably a package out there to help you

Code Portability: With most code, it can run unaltered on a plethora of computers so long as all the required modules are supplied

Large and Growing Community: People from all fields from Astronomy to Sociology are coding in Python, creating a diverse and rich community of experts all over.

WHY PYTHON?

Open Source/Free: No need to worry about licences

Cross-platform: Can be used with Windows/Macs OS/Linux

Full-featured Packages: If there's someth
probably a package out there to help y

Code Portability: With most code, it can
of computers so long as all the required

Large and Growing Community: Peopl
Astronomy to Sociology are coding in Py
rich community of experts all over.

PRO TIP:

In this workshop, we'll be using Python 2, but teach you how to code in Python 3 (which is being slowly adopted across the community)

RUNNING PYTHON

Directly from script:

```
>> python scriptname.py
```

Running a python script from beginning to end in your favourite terminal

RUNNING PYTHON

Directly from script:

```
>> python scriptname.py
```

Running a python script from beginning to e

PRO TIP:

Python scripts traditionally have the extension ".py"

RUNNING PYTHON

Directly from script:

```
>> python scriptname.py
```

Interactively:

```
>> ipython
```

Opening an “ipython” process to either run a script, or use as a “calculator” – or both!

RUNNING PYTHON

Directly from script:

```
>> python scriptname.py
```

Interactively:

```
>> ipython
```

Opening an “ipython” process to either r
“calculator” – or both!

PRO TIP:

You can also run straight python interactively, but this is not recommended

RUNNING PYTHON

Directly from script:

```
>> python scriptname.py
```

Interactively:

```
>> ipython
```

Running a script once in ipython:

```
In [1]: %run scriptname.py
```

or:

```
In [1]: execfile('scriptname.py')
```

LEAVING PYTHON

If in script: python will automatically exit when script has completed

Interactively: just type

```
In [1]: exit
```

Or press: **Ctrl-D** (on Windows, Linux, and Macs)

LEAVING PYTHON

If in script: python will automatically exit when script has completed

Interactively: just type

```
In [1]: exit
```

Or press: **Ctrl-D** (on Windows, Linux, and Macs)

PRO TIP:

Ctrl-C will not exit you out of (i)python, but rather cancel what you are currently doing

INTERACTIVE PYTHON (IPYTHON)

A special shell on top of python that makes using it interactively a breeze. It includes such features as:

- Tab-complete (both functions and variables)
- Documentation at the push of a "?"
- Full history accessible by pressing up and down
- Variables stay loaded for you to investigate and manipulate

IPYTHON MAGIC WORDS & CHARACTERS

To get documentation (for anything):

```
In [1]: functname?
```

To run a shell command:

```
In [2]: !cd dirname
```

To run a script file:

```
In [3]: %run scriptname.py
```

To time a function:

```
In [4]: %timeit command
```

To see your command history:

```
In [5]: %history
```

IPYTHON MAGIC WORDS & CHARACTERS

To get documentation (for anything):

```
In [1]: functname?
```

To run a shell command:

```
In [2]: !cd dirname
```

To run a script file:

```
In [3]: %run scriptname.py
```

To time a function:

```
In [4]: %timeit command
```

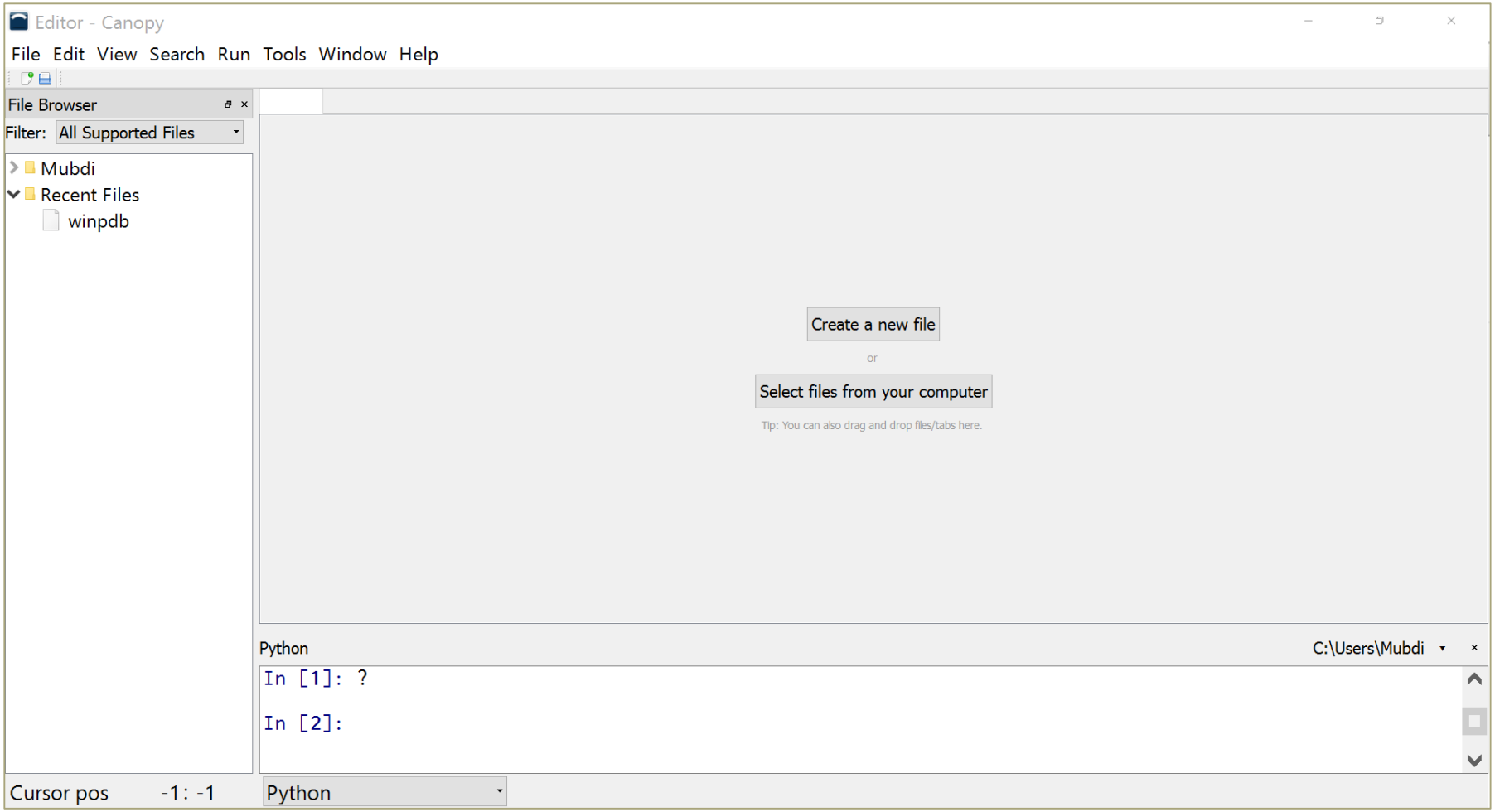
To see your command history:

```
In [5]: %history
```

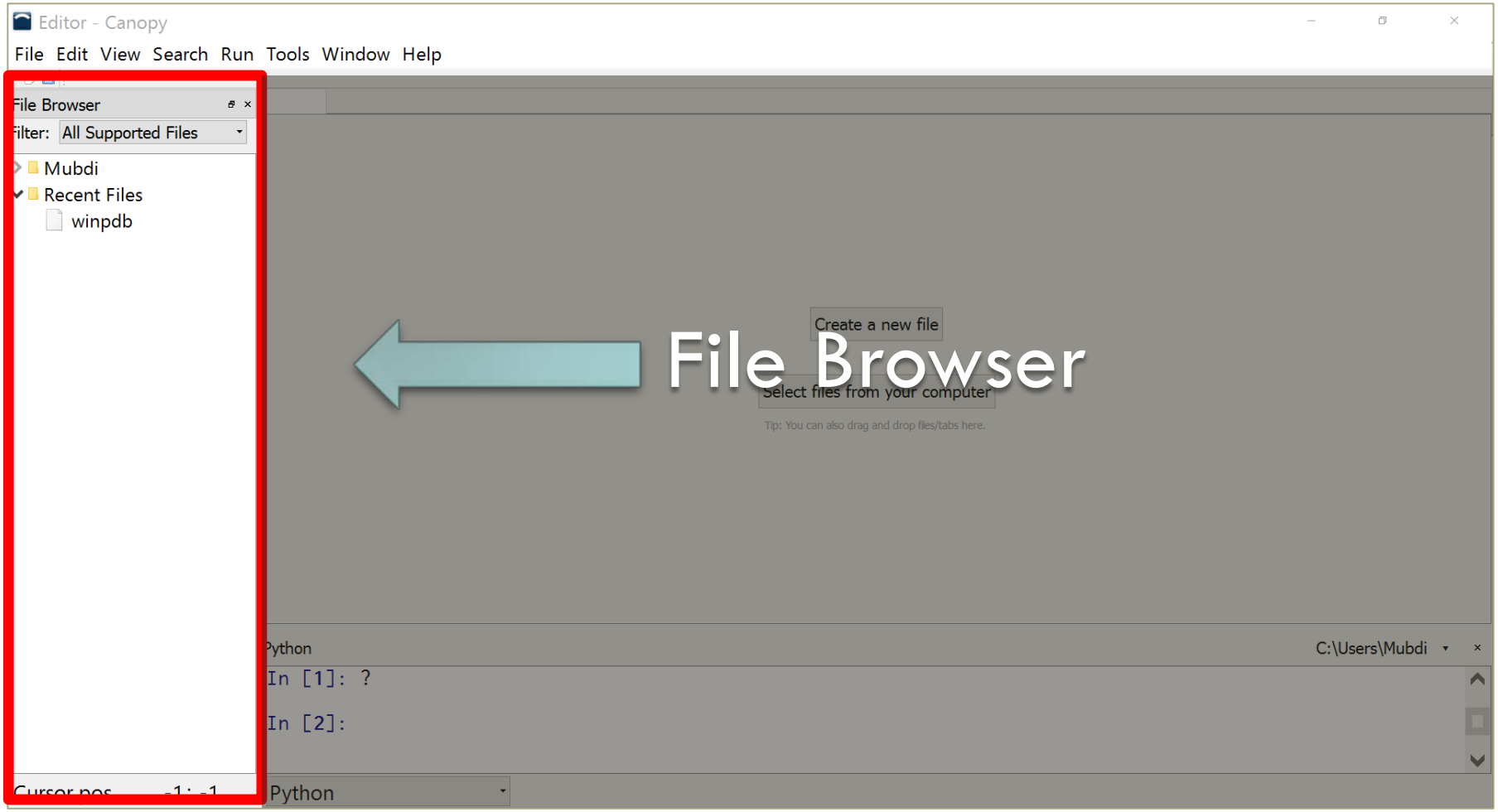
PRO TIP:

Many basic shell commands (i.e., cd, ls, pwd) work in ipython without the use of the bang (!)

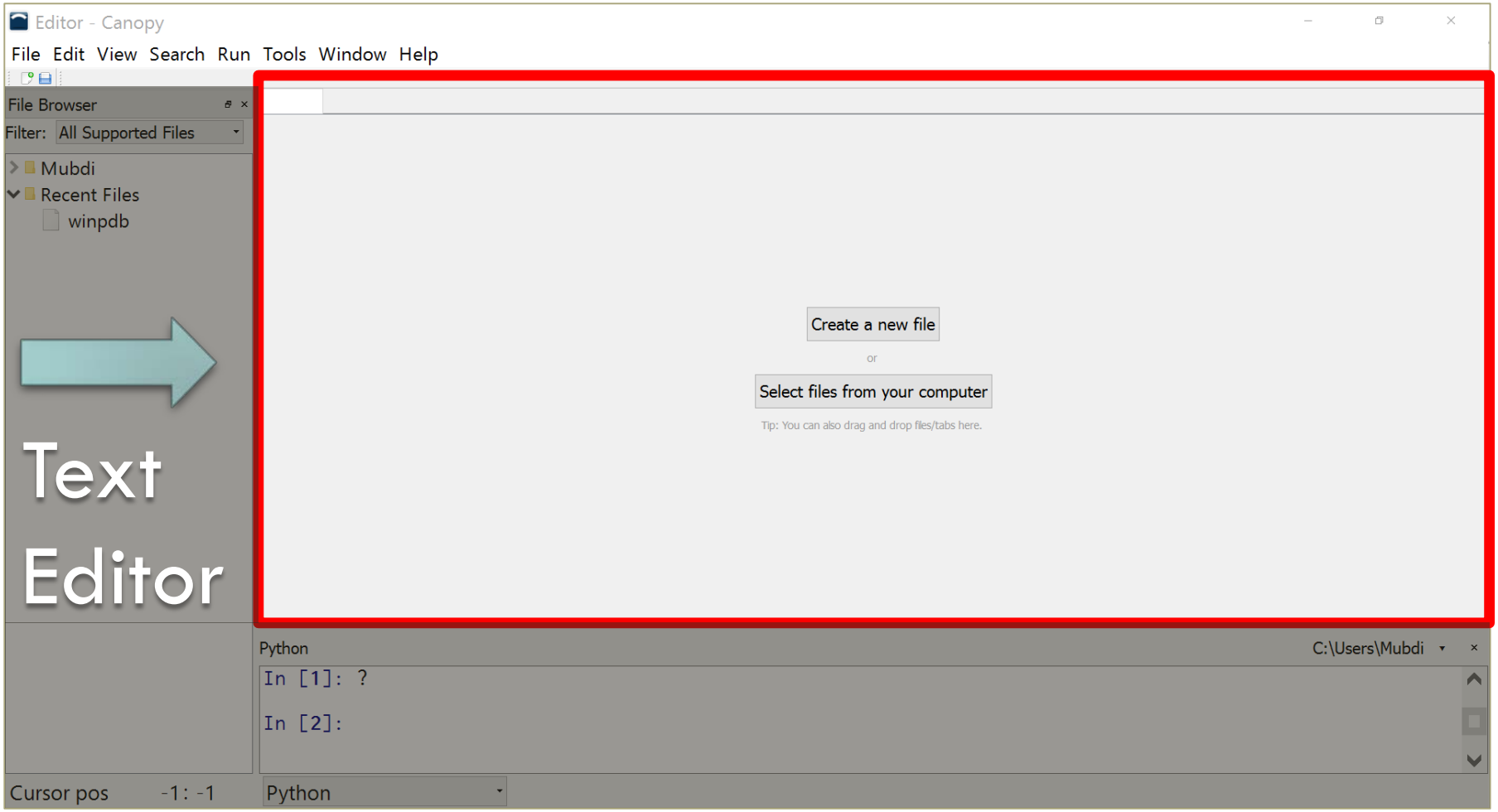
CANOPY: A USEFUL DEVELOPMENT ENVIRONMENT



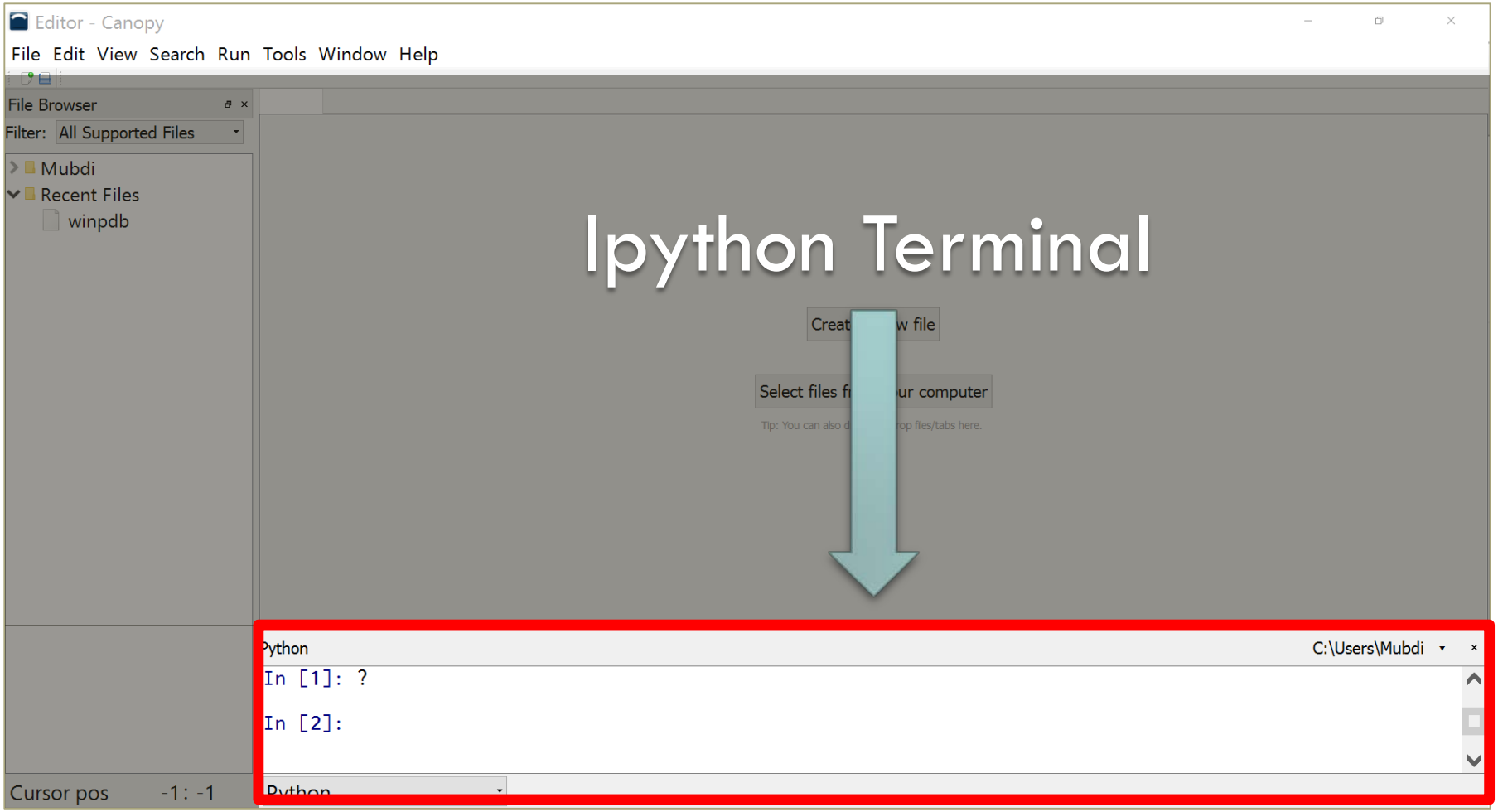
CANOPY: A USEFUL DEVELOPMENT ENVIRONMENT



CANOPY: A USEFUL DEVELOPMENT ENVIRONMENT



CANOPY: A USEFUL DEVELOPMENT ENVIRONMENT



MODULES: THE POWER OF PYTHON

The base language of Python is actually quite limited. Most of its power comes from **Modules** (or sometimes referred to as **Packages**)

Modules must be **imported** before they can be used:

```
In [1]: import module1
In [2]: import module2, module3
```

Importing single or multiple modules on a single line

Once imported, you can access functions or variables:

```
In [3]: module1.function1()
```

MODULES: THE POWER OF PYTHON

Sometimes typing the module name all the time can be annoying in which case:

Creating a shorter name or just getting the function you want

```
In [1]: import module1 as m1
In [2]: from module2 import function2
```

Once imported, you can access functions or variables:

```
In [3]: m1.function1()
In [4]: function2()
```

MODULES: THE POWER OF PYTHON

Sometimes typing the module name all the time can be annoying in which case:

Creating a shorter name or just getting the function you want

```
In [1]: impor
In [2]: from
```

Once imported, you can access functior

```
In [3]: m1.function1()
In [4]: function2()
```

PRO TIP:

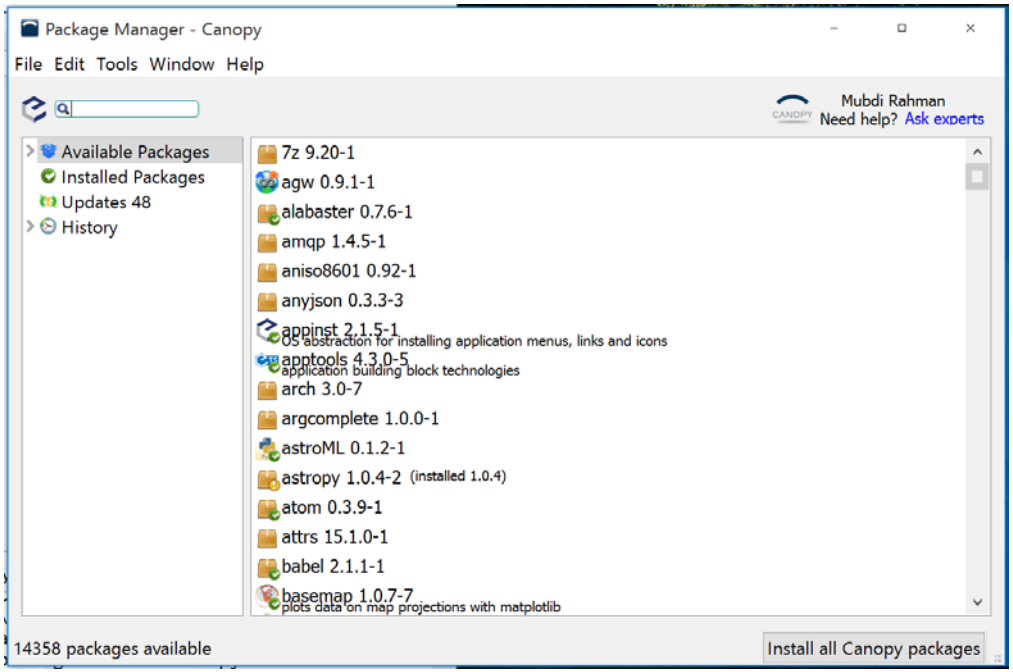
Some places will show examples that involve importing all functions in a module by:

```
from module1 import *
```

While this may seem handy, it is dangerous. **DON'T DO THIS!**

INSTALLING NEW MODULES

Enthought's Canopy provides the majority of modules you'll want and/or need automatically. But there are modules that you'll likely want to get. Canopy makes this easy using the **Package Manager**



INSTALLING NEW MODULES

Python also makes installing packages easy in general using **pip** on the command line:

```
C:\Users\Mubdi> pip install packagename
```

This downloads and installs any package available on the (centralized) Python Package Index (PyPI)

```
C:\Users\Mubdi> pip install http://url.goes.here
```

This downloads and installs the package from somewhere on the internet

BASICS OF A SCRIPT: COMMENTS

The most important part of any script

```
In [1]: # This is a comment
In [2]: # This is also a comment
```

For longer comments (in a script for instance):

```
'''
This text is in a comment
So is this text
'''
This text is outside a comment
```

BASICS OF A SCRIPT: COMMENTS

Take the comment pledge:

"I will comment liberally and consistently throughout all code I write, or so help me Python guru."

BASICS OF A SCRIPT: COMMENTS

Take the comment pledge:

"I will comment liberally and consistently throughout all code I write, or so Python g

PRO TIP:

Comment. It's the right thing to do. Just do it. Really.

BASICS OF A SCRIPT: INDENTATION

Python uses indents to indicate blocks of code – no brackets!

```
# My schematic python script

command 1
command 2
command 3
    inner command 1
    inner command 2
        more inner command 1
    inner command 3
command 4
```

BASICS OF A SCRIPT: INDENTATION

Python uses indents to indicate blocks of code – no brackets!

```
# My schematic python script

command 1
command 2
command 3
    inner command 1
    inner command 2
        more inner command
    inner command 3
command 4
```

PRO TIP:

Let your text editor deal with indenting for you. And when you need to do it yourself, use spaces not tabs.

BASICS OF A SCRIPT: VARIABLES

Variables are simple and flexible in python. There is no need to declare any variable type before setting it. And they can be set at any point throughout the script or on the fly (if using it interactively):

```
In [1]: var1 = value # No need to declare
In [2]: var2, var3 = value2, value3
In [3]: # Multiple Values can be set at once
```

Anything can be a variable in python: numbers, strings, functions, modules, *et cetera*. You can check out what type the variable is by:

```
In [3]: type(var1)
```

BASICS OF A SCRIPT: PRIMITIVE VARIABLES

There are only a few built in variables in python:

Strings (str)

Integers (int)

Floats (float)

Any form of text. These can be enclosed in single (‘) or double (“) quotes. They also accept **escape characters** (i.e., \n, \t, \a)

```
In [1]: var1 = ‘This is a String’
In [2]: var2 = “This is also a String”
```

When added, they make a longer string:

```
In [3]: var3 = var1 + var2
```

BASICS OF A SCRIPT: PRIMITIVE VARIABLES

There are only a few built in variables in python:

Strings (str)	Integers (int)	Floats (float)

Any integer (…, -1, 0, 1, …). Mathematical operations are as you expect

```
In [1]: var1, var2 = 1, 2
```

They are subject to integer math:

```
In [2]: var1/var2 # will give you 0, not 0.5
```

BASICS OF A SCRIPT: PRIMITIVE VARIABLES

There are only a few built in variables in python:

Strings (str)	Integers (int)	Floats (float)

Any integer (…, -1, 0, 1, …). Mathematical operations are as you expect

```
In [1]: var1, var2 = 1, 2
```

They are subject to integer math:

```
In [2]: var1/var2 # will give y
```

PRO TIP:

Taking an exponent uses the double asterisk character (**):

```
x = y**2
```

BASICS OF A SCRIPT: PRIMITIVE VARIABLES

There are only a few built in variables in python:

Strings (str)	Integers (int)	Floats (float)

Any integer (…, -1, 0, 1, …). Mathematical operations are as you expect

```
In [1]: var1, var2 = 1, 2
```

They are subject to integer math:

```
In [2]: var1/var2 # will give yo
```

PRO TIP 2:

This is the only case throughout this workshop that something will change in Python 3: all math will default to floating point math.

BASICS OF A SCRIPT: PRIMITIVE VARIABLES

There are only a few built in variables in python:

Strings (str) Integers (int) Floats (float)

Any real number (1.0, 2.5, 1e25). Mathematical operations are as you expect

```
In [1]: var1, var2 = 1.0, 2e2
```

Any operation with an int and a float will give you a float:

```
In [2]: 1/2.0 # will give you 0.5, not 0
```

BASICS OF A SCRIPT: PRIMITIVE VARIABLES

There are only a few built in variables in python:

Strings (str)	Integers (int)	Floats (float)

Any real number (1.0, 2.5, 1e25). Mathematical c

```
In [1]: var1, var2 = 1.0, 2e2
```

Any operation with an int and a float will give yo

```
In [2]: 1/2.0 # will give you 0.
```

PRO TIP:

Every variable in python is an object that has methods (functions) associated with it. You can access these with the dot character (.) after the variable name:

```
var1.method()
```

BASICS OF A SCRIPT: LISTS

Basic ordered grouping of any type of variables:

```
In [1]: list1 = [1, 2, 3]
# Lists can contain different types of variables
In [2]: list2 = [1, ‘a’, 3.4]
# You can make lists of lists
In [3]: list3 = [1, [‘a’, ‘b’, ‘c’] , 3.4]
```

Accessing individual components of a list:

```
In [4]: x = list1[2] # Returns the 3rd element
# Indexing of lists starts at 0 and goes to n-1
In [5]: len(list1) # Gets the size of the list
```

BASICS OF A SCRIPT: LISTS

Useful functions associated with lists:

```
# Create a list of integers from 0 to 3
In [1]: list1 = range(4)

# Sort your list
In [2]: sortedlist = sort(list2)

# Add more to your list
In [3]: list3 = list3.append(newvariable)

# Combine multiple lists together
In [4]: combinedlist = list1 + list2
```

BASICS OF A SCRIPT: LISTS

Useful functions associated with lists:

```
# Create a list of integers from 0 to 3
In [1]: list1 = range(4)

# Sort your list
In [2]: sortedlist = sort(list2)

# Add more to your list
In [3]: list3 = list3.append(newvariable)

# Combine multiple lists together
In [4]: combinedlist = list1 + l
```

PRO TIP:

You can create an empty list to append to:

```
emptylist = []
```

BASICS OF A SCRIPT: TUPLES

Ordered grouping of variables. Not as flexible as lists (not *mutable*) but the basics are the same:

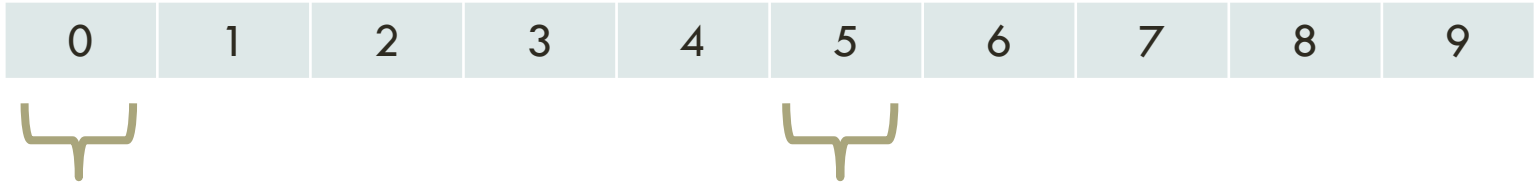
```
In [1]: tuple1 = (1, 2, 3)
In [2]: tuple2 = (1, ‘a’, 3.4)
In [3]: tuple3 = (1, (‘a’, ‘b’, ‘c’), 3.4)
```

Can also quickly assign values from within tuples:

```
In [4]: tuple4 = (1, 2, 3)
In [5]: var1, var2, var3 = tuple4
# also works for lists
```

BASICS OF A SCRIPT: INDEXING

Taking a simple list:




```
list[0]
```

```
list[5]
```

Simple indexing

BASICS OF A SCRIPT: INDEXING

Taking a simple list:

0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

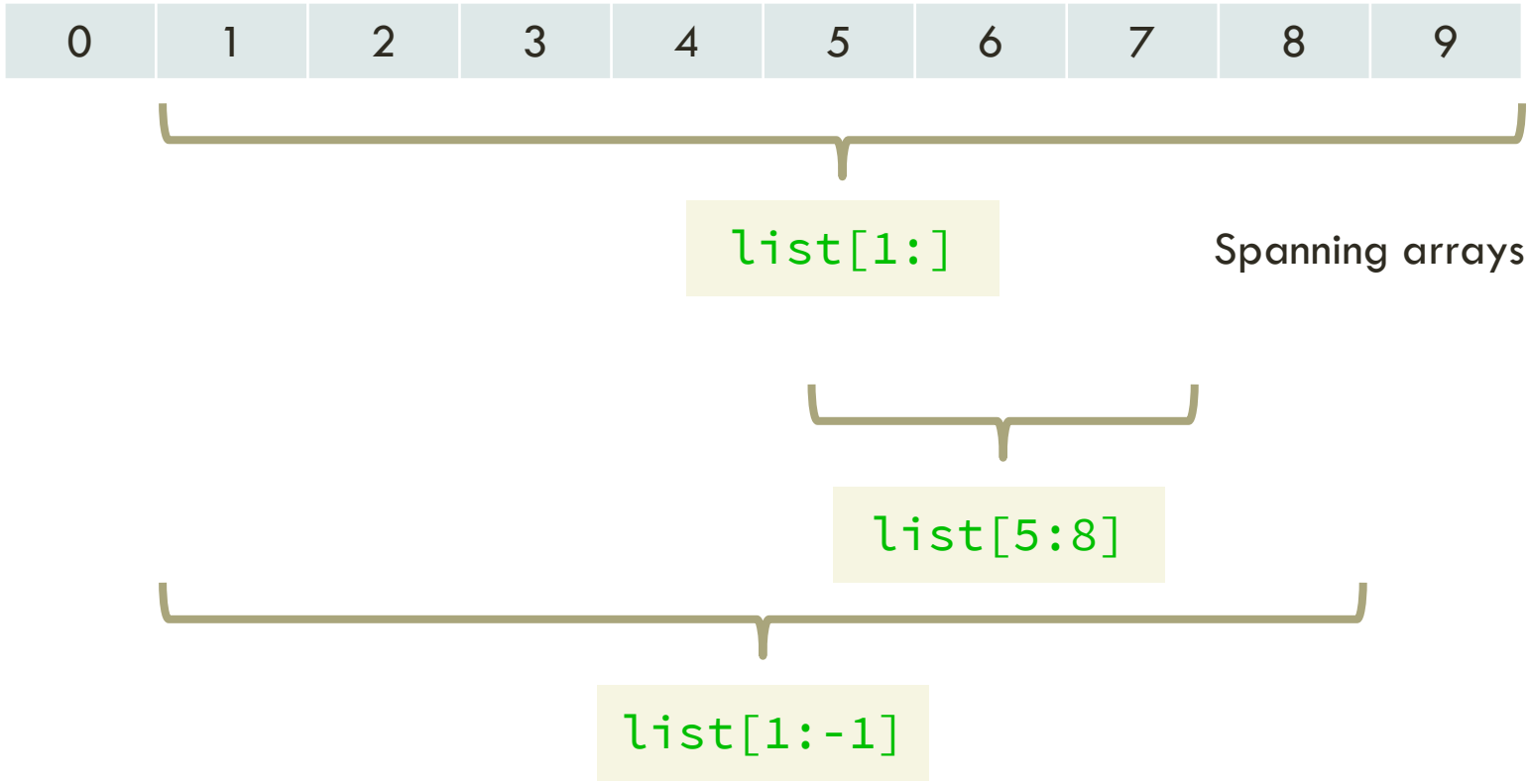
```
list[-10]
```

```
list[-5]
```

Inverse indexing

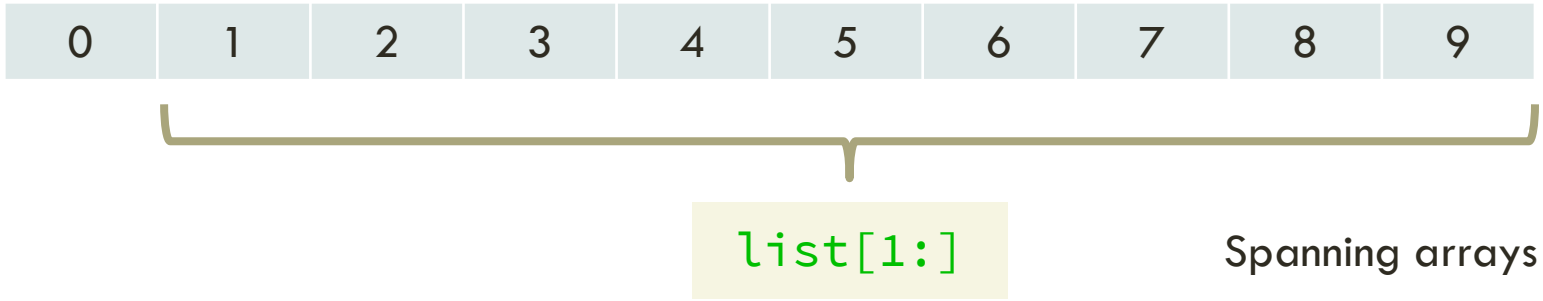
BASICS OF A SCRIPT: INDEXING

Taking a simple list:



BASICS OF A SCRIPT: INDEXING

Taking a simple list:

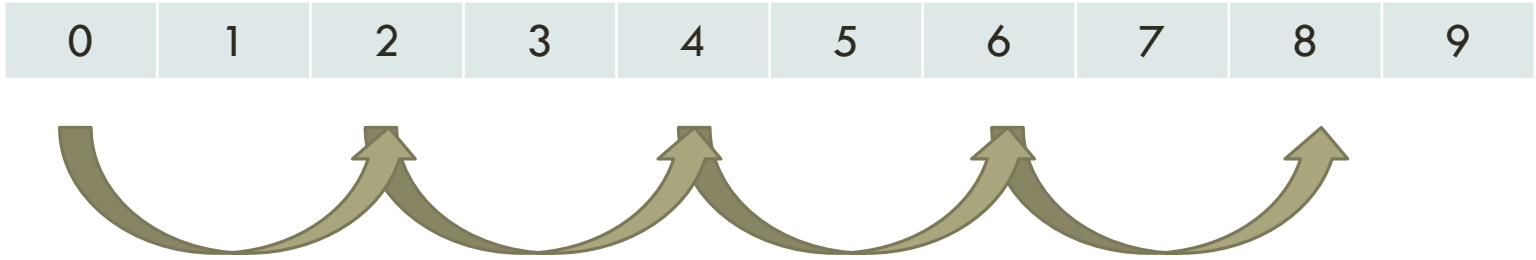


PRO TIP:

The range you choose will exclude the value at the stop number

BASICS OF A SCRIPT: INDEXING

Taking a simple list:

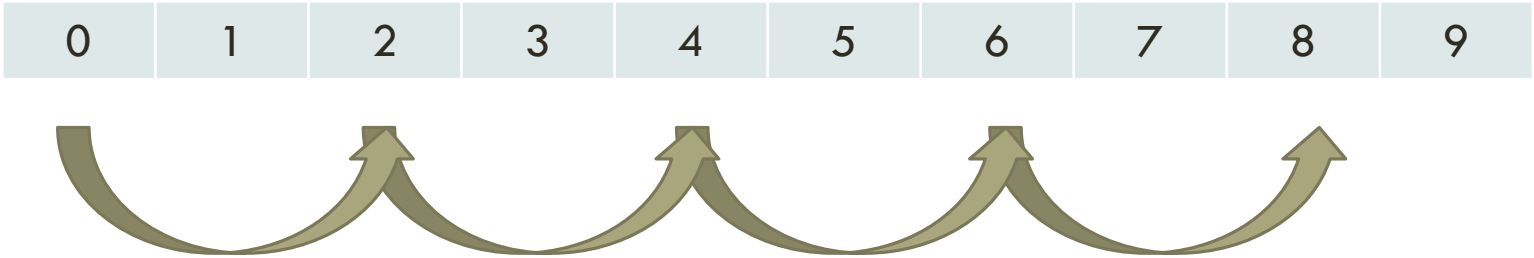



```
list[::2]
```

Changing counting

BASICS OF A SCRIPT: INDEXING

Taking a simple list:




```
list[::2]
```

Changing counting

PRO TIP:

To reverse the order of an array, just use:

```
[::-1]
```

BASICS OF A SCRIPT: DICTIONARIES

Unordered grouping of variables accessed by key. Anything can be a *key* or a *value*:

```
In [1]: dict1 = {‘val1’:1, 2:2, ‘val3’:3}
In [2]: dict1[‘val1’] # Returns 1
In [3]: dict1[‘newval’] = 4 # Add new value
```

Can quickly get all the keys in a dictionary (in a list):

```
In [4]: dict1keys = dict1.keys()
```

BASICS OF A SCRIPT: FUNCTIONS

Making a function is quite simple and can be defined anywhere:

```
def function1(var1, var2):
    # Your Code goes here
    var3 = var1 + var2
    return var3
```

You can define optional arguments and return multiple values:

```
def function1(var1, var2='value'):
    var3 = var1 + var2
    return var3, var2 # This will be a tuple
```

BASICS OF A SCRIPT: FLOW CONTROL

Conditional (if-else) statements:

```
if var1 == 0:
    Code to run if var1 is 0
elif var1 == 1:
    Code to run if var1 is 1
else:
    Code to run otherwise
```

BASICS OF A SCRIPT: FLOW CONTROL

While loop:

```
while var1 > 5:
    Code to run (in a loop) if var1 > 5
```

For loop:

```
for tmp_var in list1:
    tmp_var is set to the values in list1

# If you want a for loop with numbers from 0 to N-1:

for tmp_var in range(N):
    Code to run with tmp_var = 1 to N
```

BASICS OF A SCRIPT: FLOW CONTROL

Special keywords when you are within loops or conditionals

Skip everything else in this iteration and move to the next:

```
continue
```

Exit out of the most recent loop:

```
break
```

These keywords are usually put in conjuction with an if statement.

BASICS OF A SCRIPT: PRINTING

Printing is very easy:

```
In [1]: print(“This is a message.”)
In [2]: print(variable1)
```

Mixing variables and text is also easy:

```
In [1]: print(“This is a %s message.” % “string”)
In [2]: print(“%f, %i” % (2.0, 2)) # Using tuple
In [3]: print(“%1.3f” % 1.12345) # Prints 1.123
```

EXERCISE TIME!

C is for cookie. That's good enough for me.