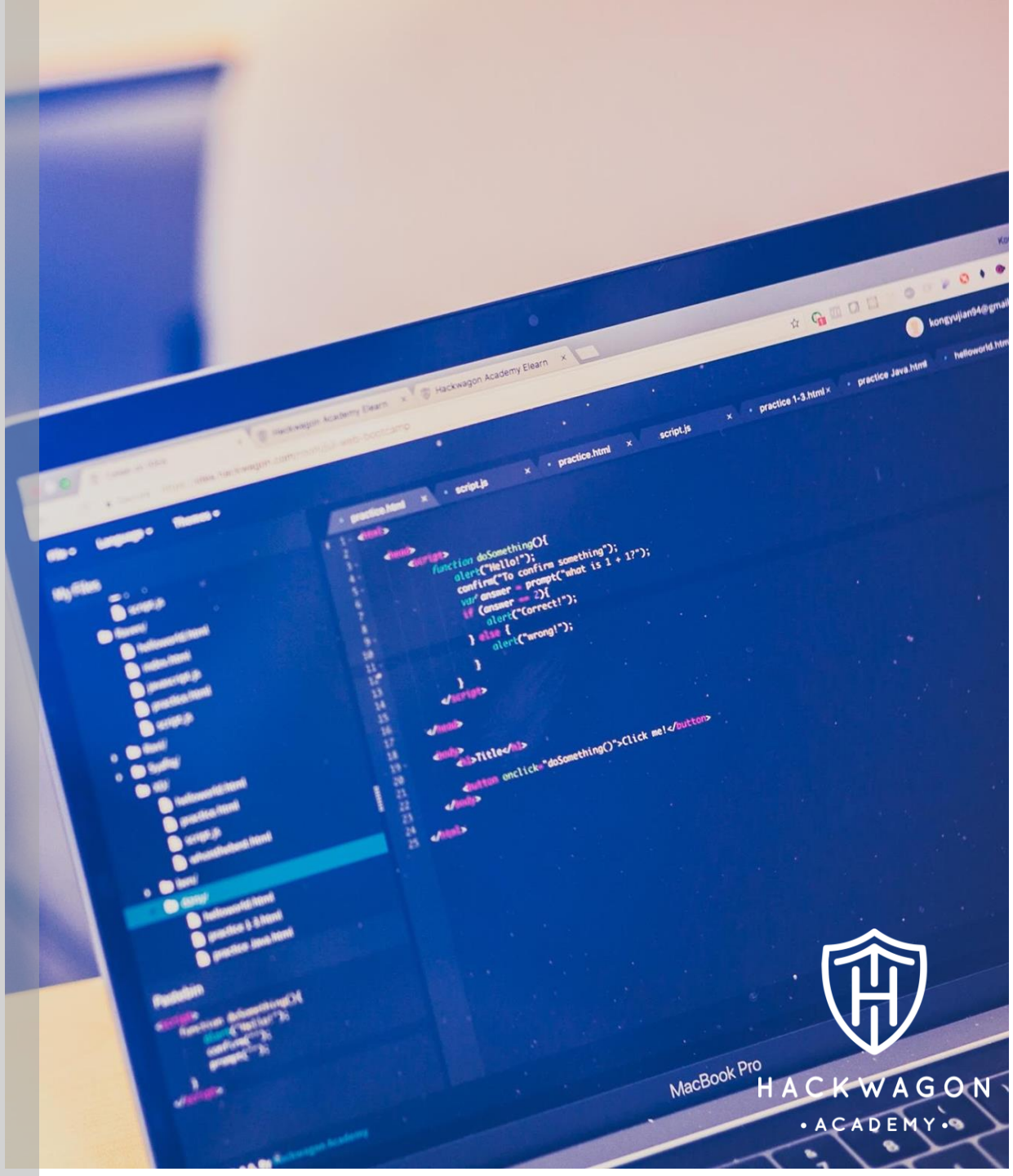
DATA SCIENCE 101:
PYTHON EXPRESSIONS AND VARIABLES

AGENDA

- Setting up development environment / Anaconda
- Why Python?
- Basic Python operations
- Variables
- Print & format functions
- Art of debugging
- Summary

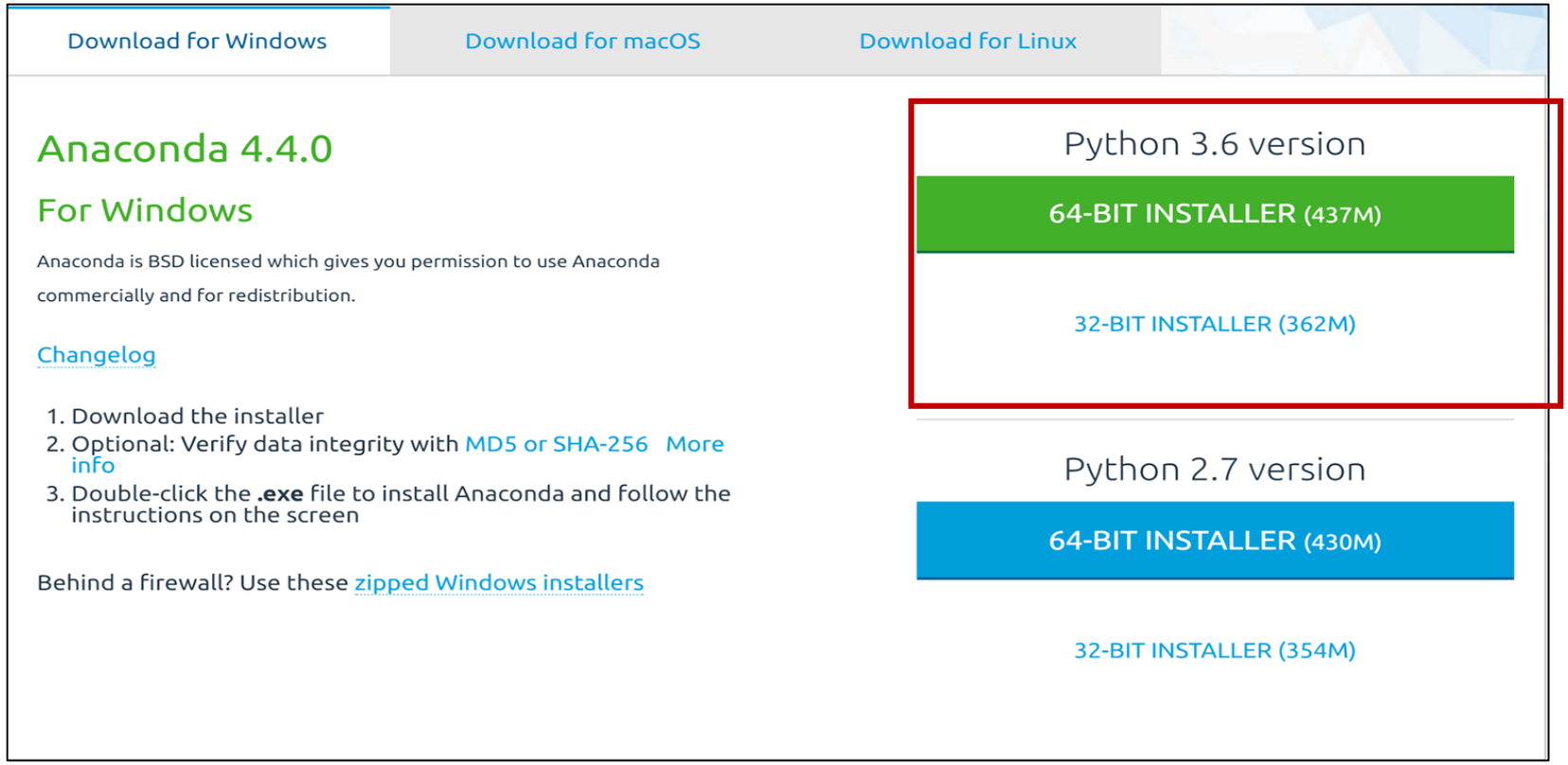


SETTING UP THE DEVELOPMENT ENVIRONMENT

- Step 1: Go to the link https://www.continuum.io/downloads

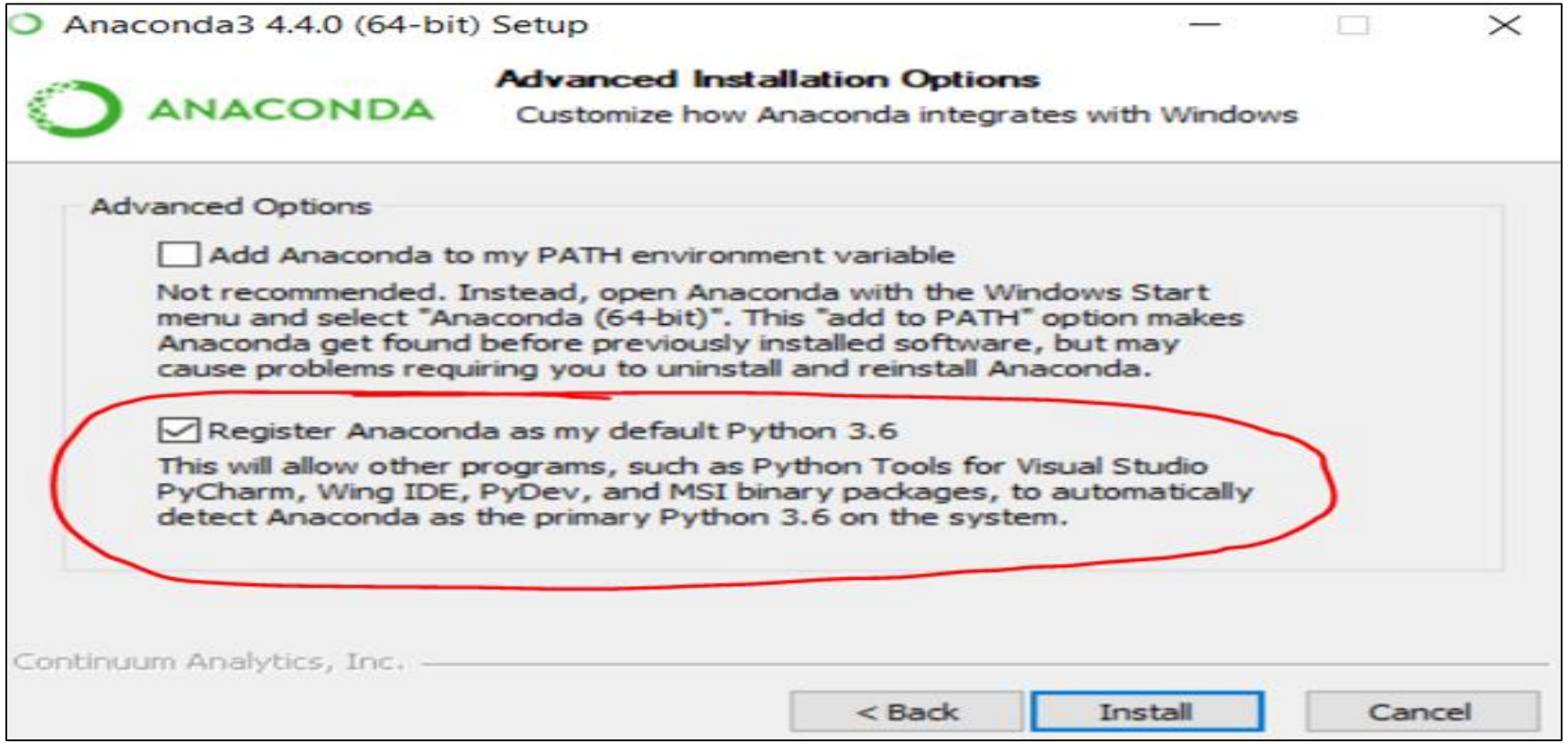
SETTING UP THE DEVELOPMENT ENVIRONMENT

- Step 2: Scroll down, and select the 64-bit installer (Python 3.6 version) if your machine runs on 64-bit, or alternatively select the 32-bit installer (Python 3.6 version) if your machine runs on 32-bit.



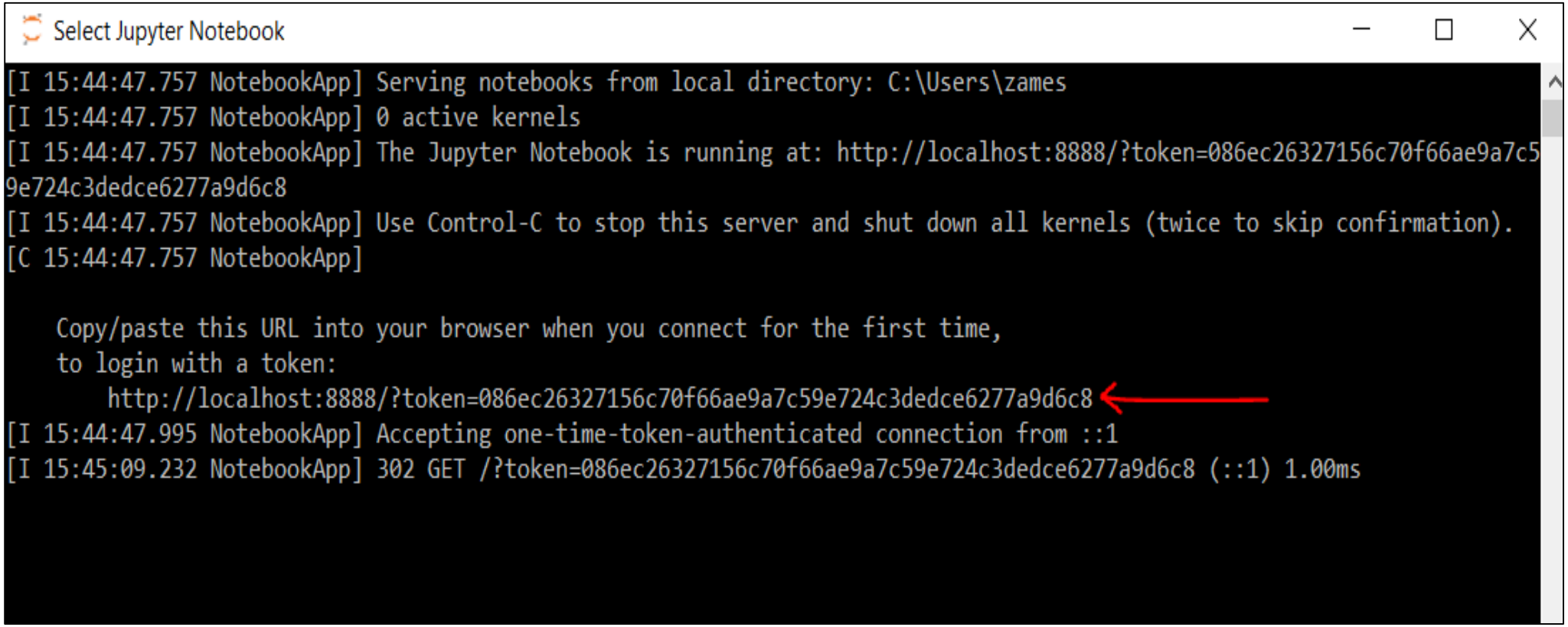
SETTING UP THE DEVELOPMENT ENVIRONMENT

- Step 3: You can follow the recommended settings during the installation, but for Advanced Installation Options, make sure to ‘Register Anaconda as my default Python 3.6’



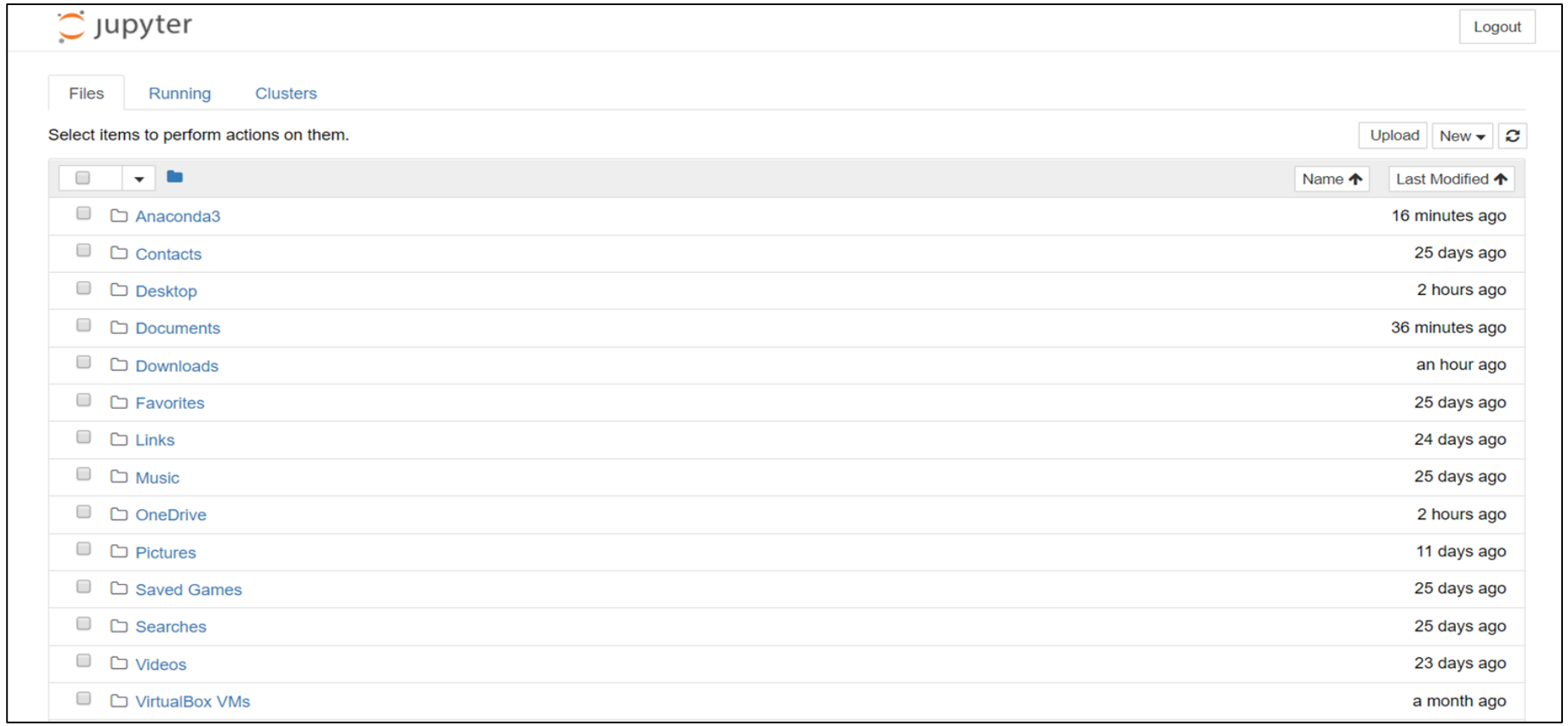
SETTING UP THE DEVELOPMENT ENVIRONMENT

- Step 4: When the installation is complete, you should be able to access an application called ‘Jupyter Notebook’. Open the application, wait for it to start up, copy the link provided and paste it into your internet browser.



SETTING UP THE DEVELOPMENT ENVIRONMENT

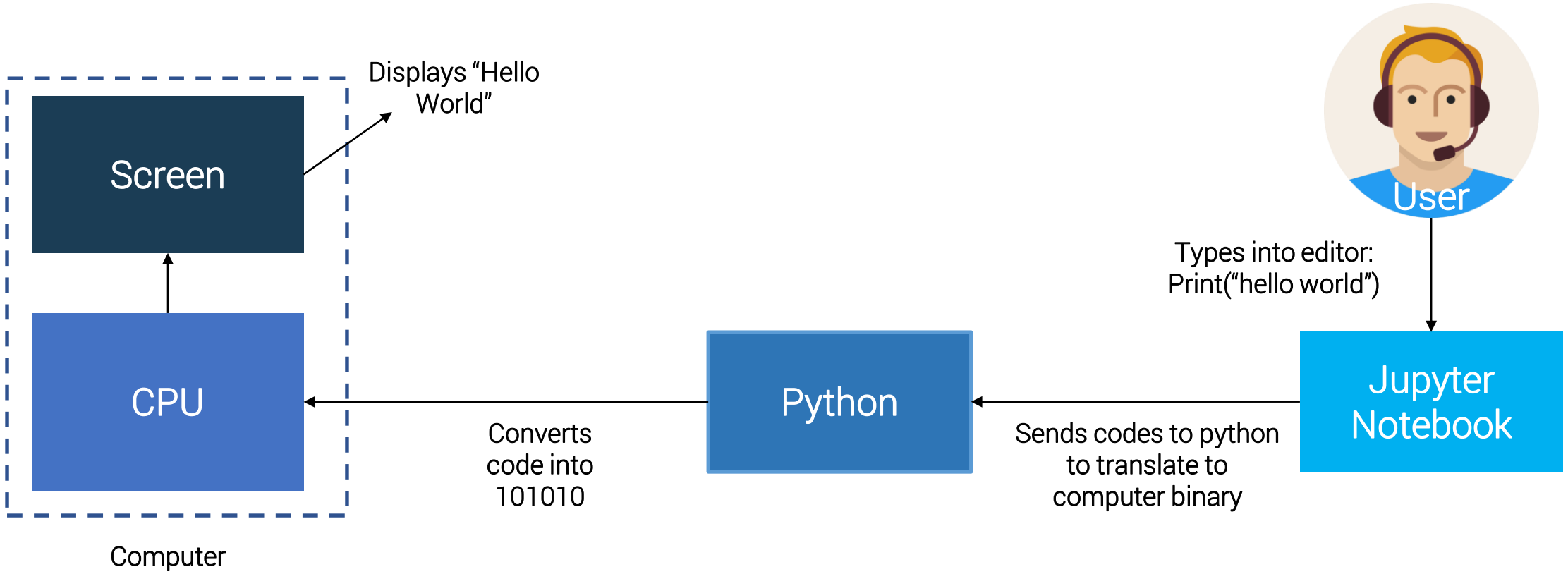
- Step 5: On your web browser, you should be able to see the Jupyter Notebook interface. If you're able to arrive at this screen, the installation should have been completed successfully!



ANACONDA – TYING EVERYTHING TOGETHER

- A software package which consists of both Python and Jupyter notebook. Hence, once we install Anaconda, we essentially have both Python and Jupyter notebook installed on our machines.

HARDWARE, PYTHON & JUPYTER NOTEBOOK



WHY PYTHON?

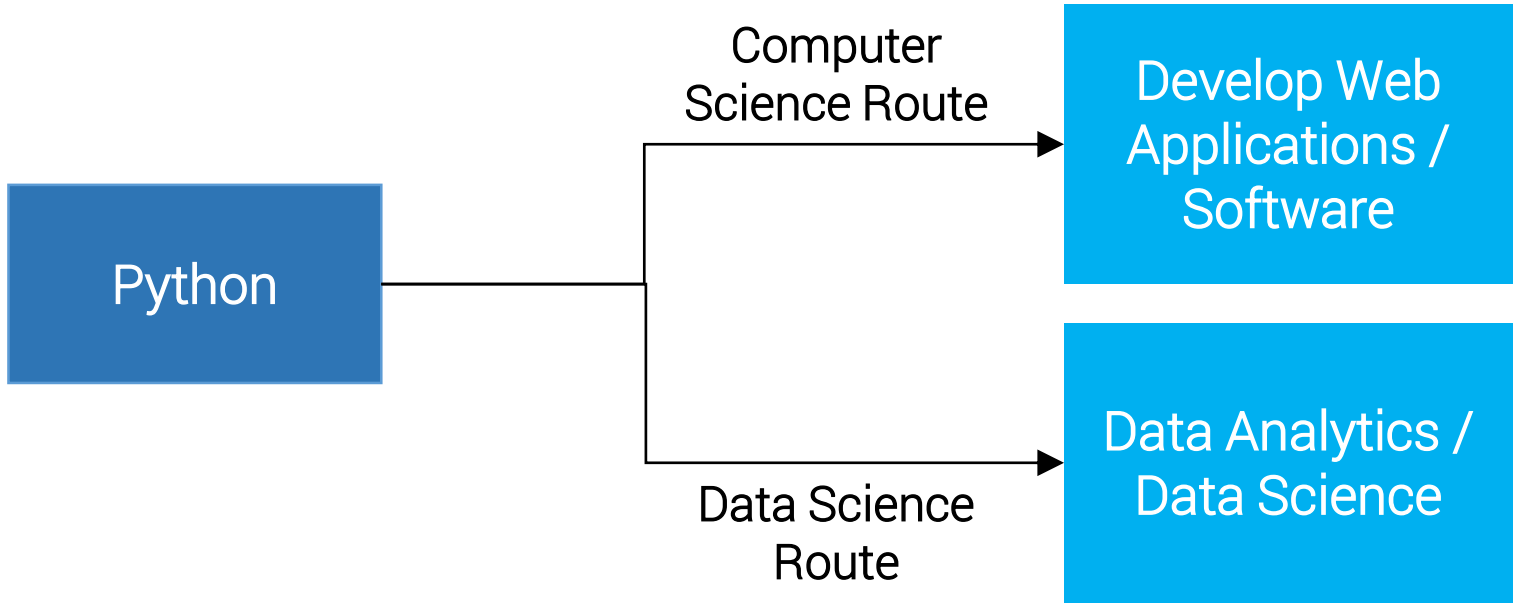
- Background information of Python
- Why learn Python?

BACKGROUND INFORMATION OF PYTHON

- A programming language developed in 1991 by Guido Van Rossum
- “Over six years ago, in December 1989, *I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas.”*

WHY LEARN PYTHON?

- Named as the most in-demand coding language in America
- One of the most favourite language used by data scientists
- Being a multi-purpose language, one can build application via python on top of being able to perform data analysis,

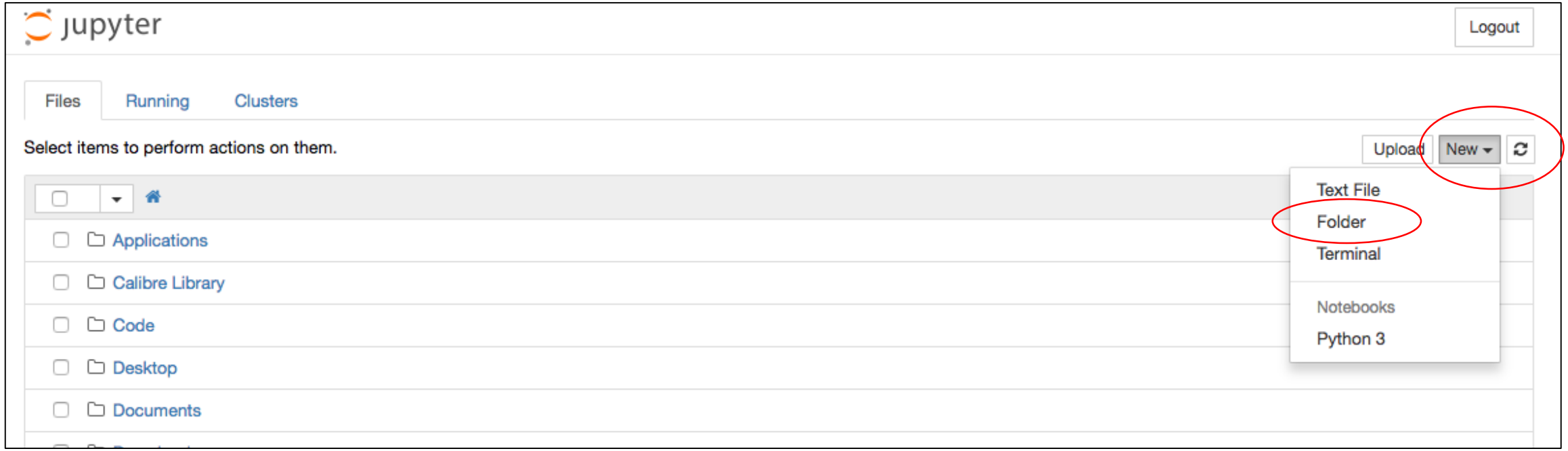


BASIC PYTHON OPERATIONS

- Getting started with Jupyter Notebook
- Print “hello world!”
- Try out the arithmetic operations in Python

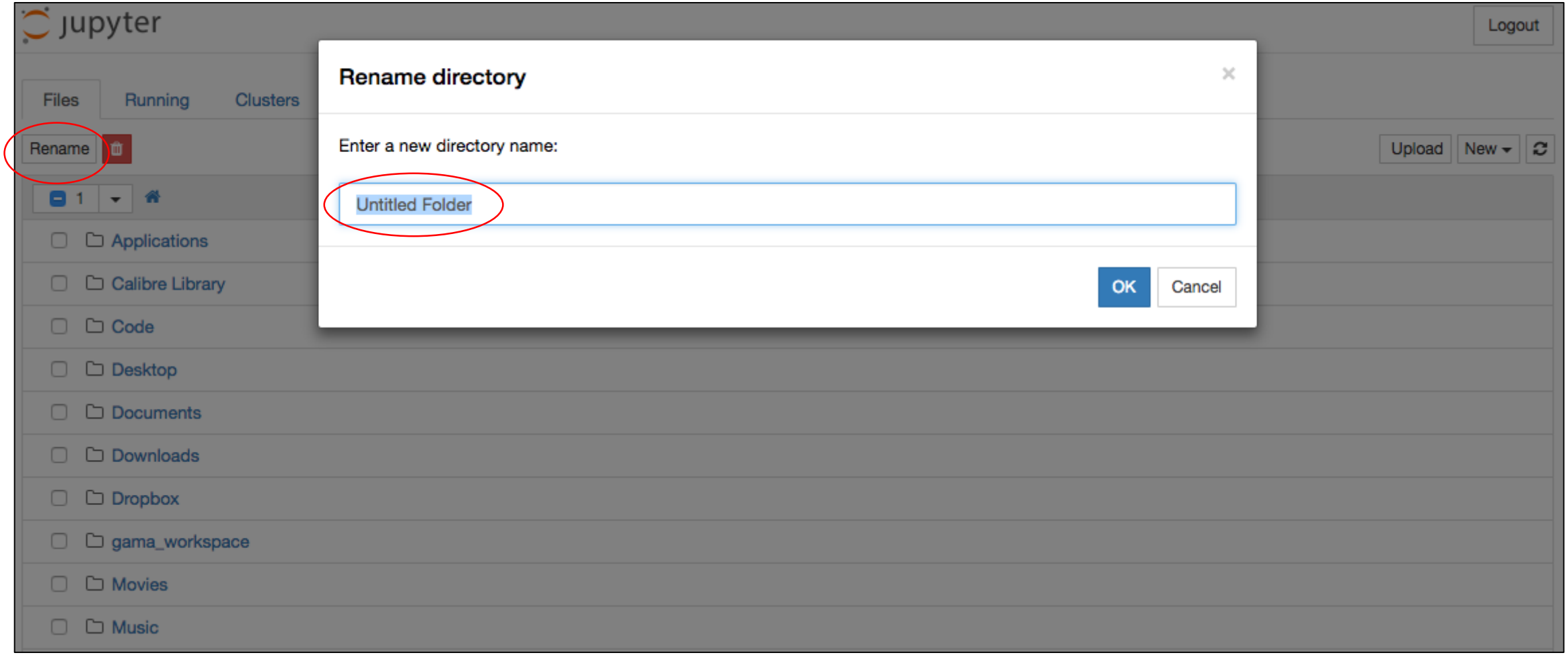
GETTING STARTED WITH JUPYTER NOTEBOOK*

- Create a new folder



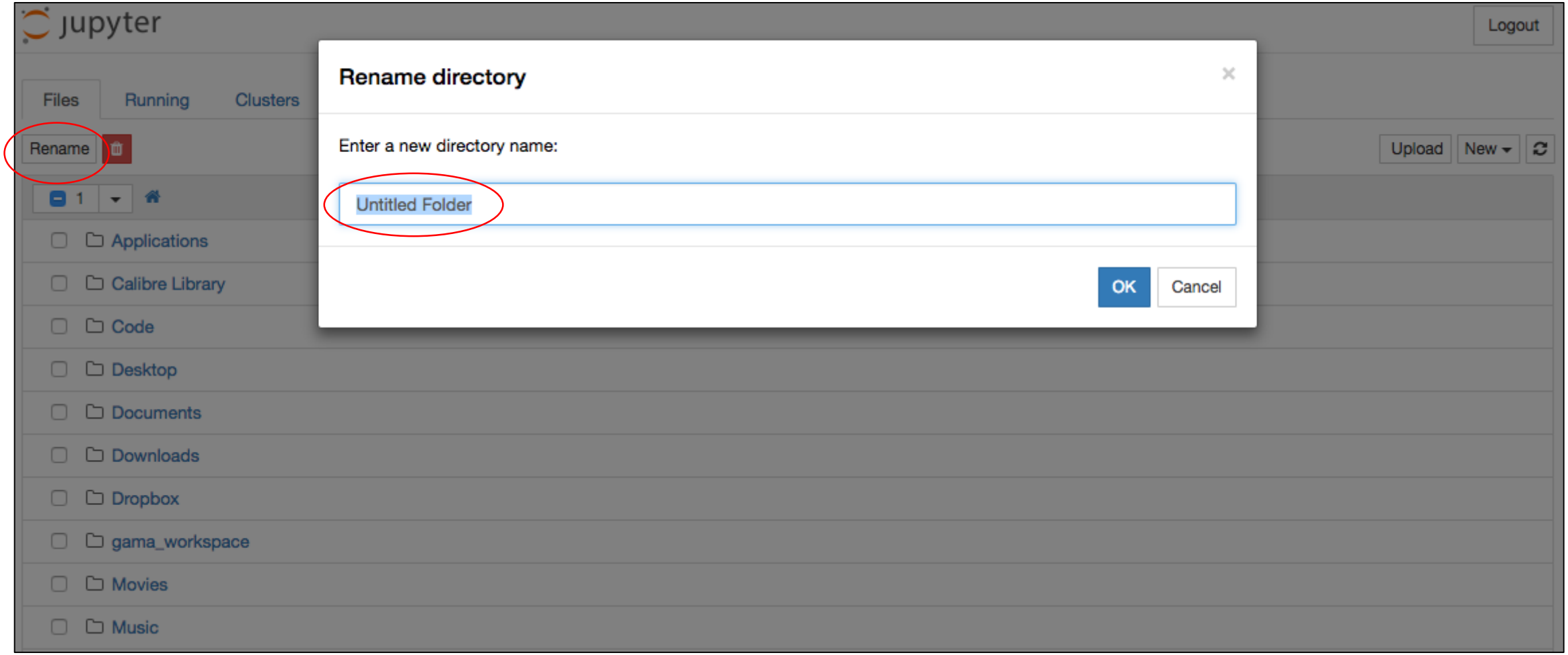
GETTING STARTED WITH JUPYTER NOTEBOOK*

- Rename folder to “Hackwagon”



GETTING STARTED WITH JUPYTER NOTEBOOK*

- Within the “Hackwagon” folder, create another folder call “Data Science 101”

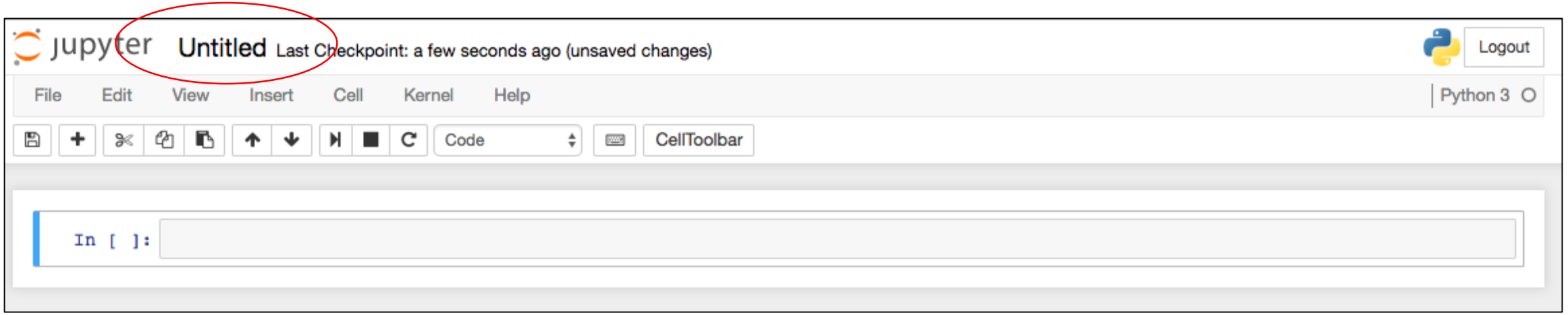


GETTING STARTED WITH JUPYTER NOTEBOOK*

- Create your new Python file in your “Data Science 101” folder

GETTING STARTED WITH JUPYTER NOTEBOOK*

- Rename the file to “In-Class Exercise Day 1”



PRINT “HELLO WORLD!”*

- Within a cell, type `print(“Hello World!”)` and click on the run button
- You should see the program output `“Hello World”`

IN-CLASS PRACTICE: ARITHMETIC OPERATIONS*

- The following are some of the basic mathematical operations in python. Try them yourselves and we'll go through how they work together.

  - print(2*3)
  - print(2**3)
  - print(8/3)
  - print(8//3)
  - print(8%2)
  - print(8%3)

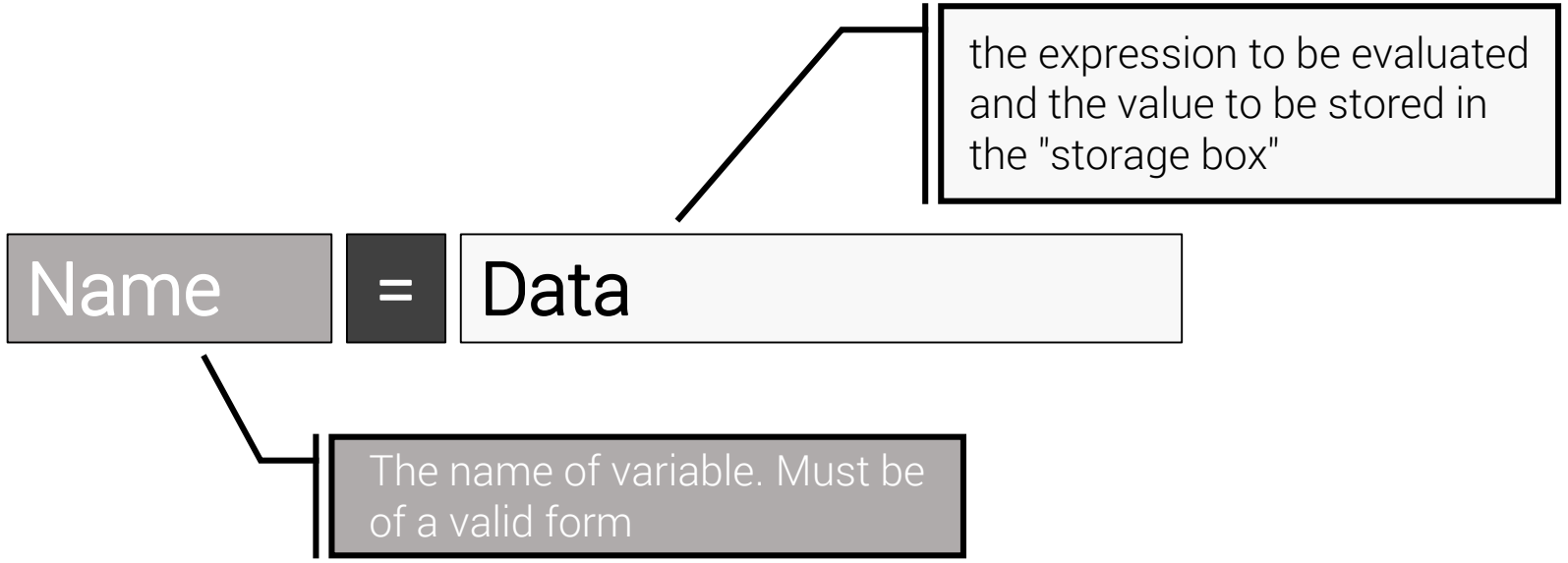
VARIABLES

- What is a variable?
- Rules for variables
- Data types
- Conversion function
- User input
- Primitive vs reference variables
- How are primitive variables stored?
- How are reference variables stored?

WHAT IS A VARIABLE?

- A variable is a named place in the memory where a programmer can store data and later retrieve the data using the variable name

- Programmers get to choose the names of the variables



RULES FOR VARIABLES: NAMING CONVENTION

- Variables names must start with a letter or an underscore, such as:
  - _underscore
  - underscore_

- The remainder of your variable name may consist of letters, numbers and underscores:
  - password1
  - N00b
  - Hello_123

- Variable names are case sensitive:
  - case_sensitive,
  - CASE_SENSITIVE, and
  - Case_Sensitive are each a different variable

RULES FOR VARIABLES: MNEMONIC VARIABLE NAMES

- Since we programmers are given a choice in how we choose our variable names, there is a bit of “best practice”
- We name variables to help us remember what we intend to store in them (“mnemonic” = “memory aid”)
- This can confuse beginning students because well named variables often “sound” so good that they must be keywords

EXAMPLE 1	EXAMPLE 2	EXAMPLE 3

`x1q3z9ocd = 35.0`
`x1q3z9afd = 12.50`
`x1q3p9afd = x1q3z9ocd * x1q3z9afd`
`print(x1q3p9afd)`
	`hours = 35.0`
	`rate = 12.50`
	`pay = hours * rate`
	`print(pay)`
		`a = 35.0`
		`b = 12.50`
		`c = a * b`
		`print(c)`

RULES FOR VARIABLES: RESERVED KEYWORDS

- You cannot use reserved keywords as variable names:

```
      and   del   for   is   raise
 assert   elif   from   lambda   return
   break   else   global   not   try
    class   except   if   or   while
continue   exec   import   pass   yield
       def   finally   in   print
```

RULES FOR VARIABLES: ASSESSING THE CONTENT

```
video_like = 40
print(video_like)
```

Output:
40

- Basically, think of variables as a symbolic name for your container. When you call a container, you assess the contents within the container.
- So you can name your container in any name which you like as long as it fits the python syntax convention.

RULES FOR VARIABLES: REASSIGNMENT

```
video_like = 40
video_like = 80
print(video_like)
```

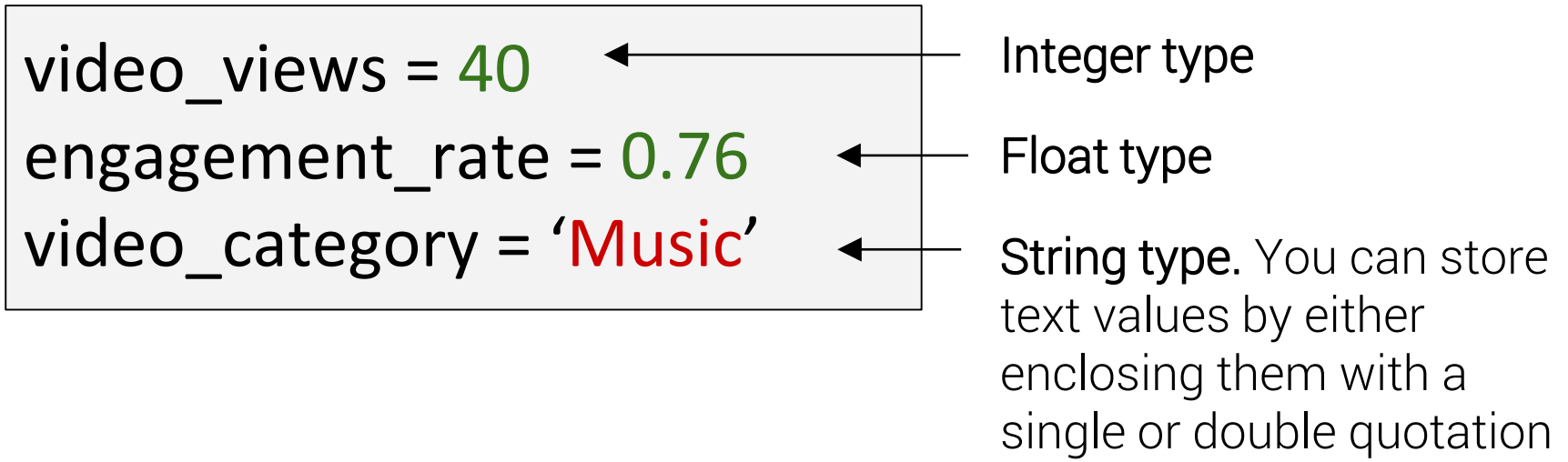
Output:
80

- You can change the contents of a variable in a later statement by simply reassigning a new value to it

DATA TYPES: IN PYTHON

- Python needs to know how to set aside memory in your computer based on the kind of information you want to store
- There are 4 basic types of data which we will be working with throughout this course
  - Numeric (Integer & Float)
  - Strings (Character-based data)
  - Logical Values (True/False)
  - Collections (List, Tuple & Dictionary)

DATA TYPES: DYNAMIC TYPING

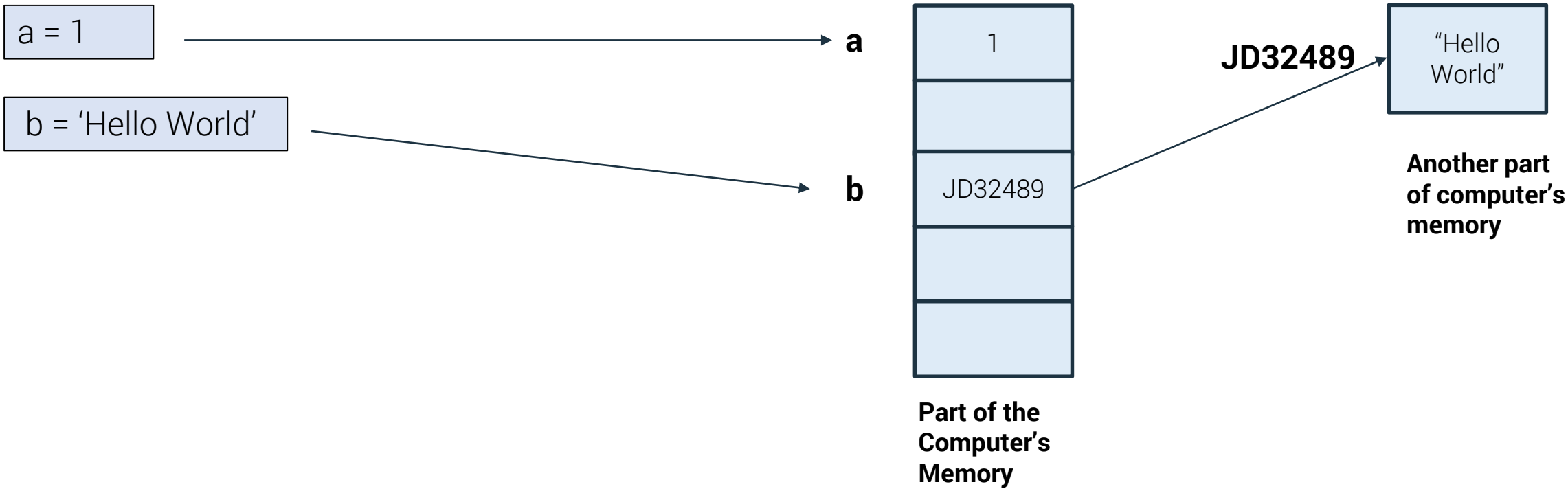


- Notice that you didn't have to do anything special to make a variable to be of a certain data type
- Python is a dynamically-typed language. This means you don't have to declare what kind of data your variables will be holding

PRIMITIVE VS REFERENCE VARIABLES

- While all variables are dynamically typed, and hence the difference between the different data types are transparent to you, it still helps to have an understanding of what goes behind the scenes

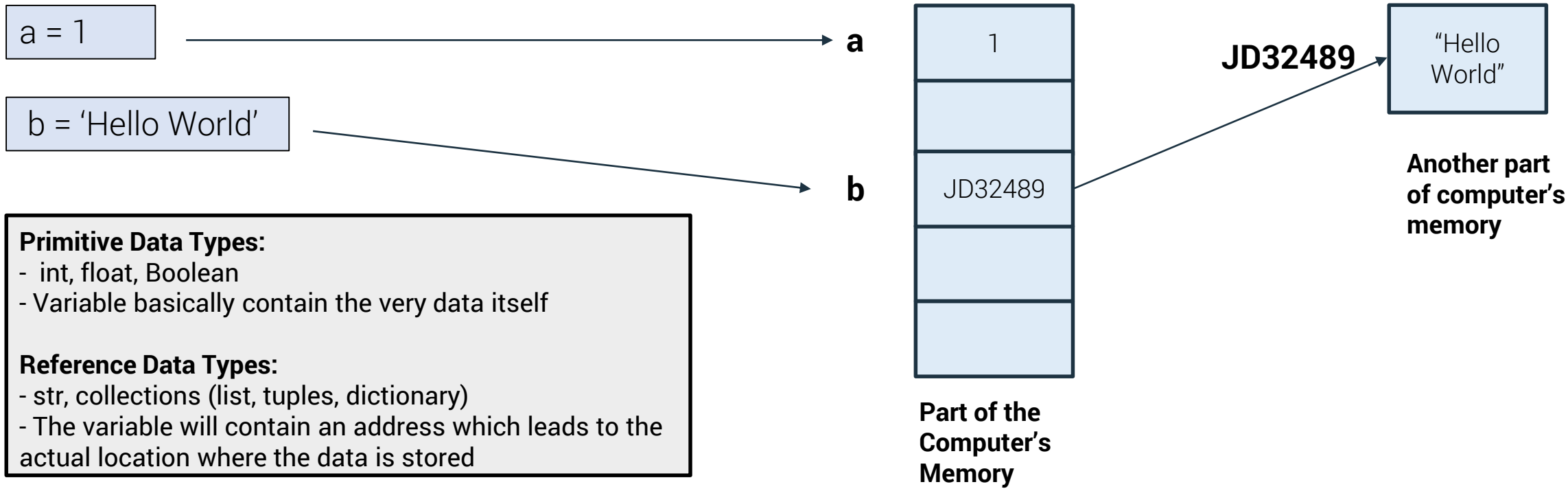
Imagine I have the two lines of codes below, an int and str type:



PRIMITIVE VS REFERENCE VARIABLES

- While all variables are dynamically typed, and hence the difference between the different data types are transparent to you, it still helps to have an understanding of what goes behind the scenes

Imagine I have the two lines of codes below, an int and str type:



Primitive Data Types:
- int, float, Boolean
- Variable basically contain the very data itself

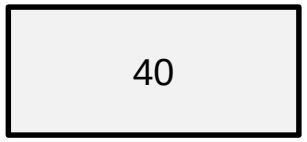
Reference Data Types:
- str, collections (list, tuples, dictionary)
- The variable will contain an address which leads to the actual location where the data is stored

HOW ARE PRIMITIVE VARIABLES STORED?

- What happens in the memory states of CPU:

```
video_views = 40
engagement_rate = 0.76
video_views = 70
```

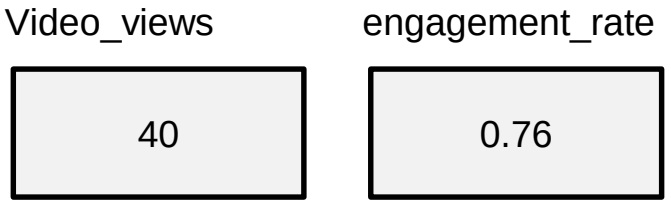
Video_views



HOW ARE PRIMITIVE VARIABLES STORED?

- What happens in the memory states of CPU:

```
video_views = 40
engagement_rate = 0.76
video_views = 70
```

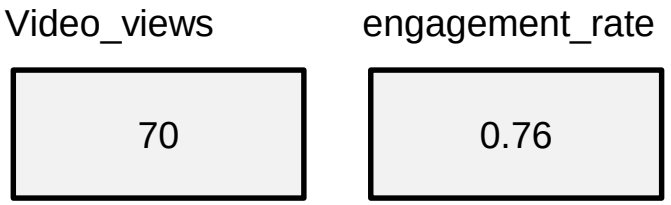


HOW ARE PRIMITIVE VARIABLES STORED?

- What happens in the memory states of CPU:

```
video_views = 40
engagement_rate = 0.76
video_views = 70
```

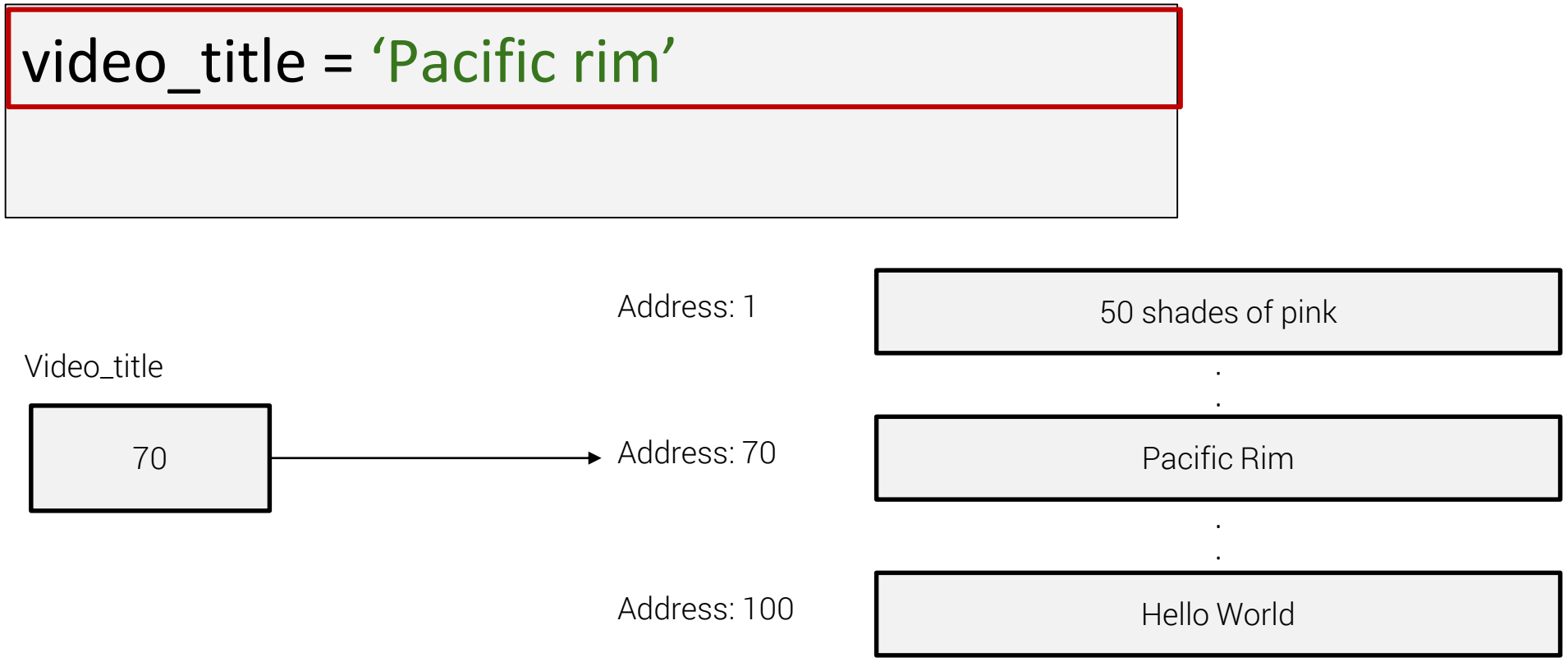
```
print(engagement_rate)
```



Python will grab the data of the variable with the name in the memory space, and return it to be printed. That's why when you call the name of the variable you are just getting the data behind it!

HOW ARE REFERENCE VARIABLES STORED?

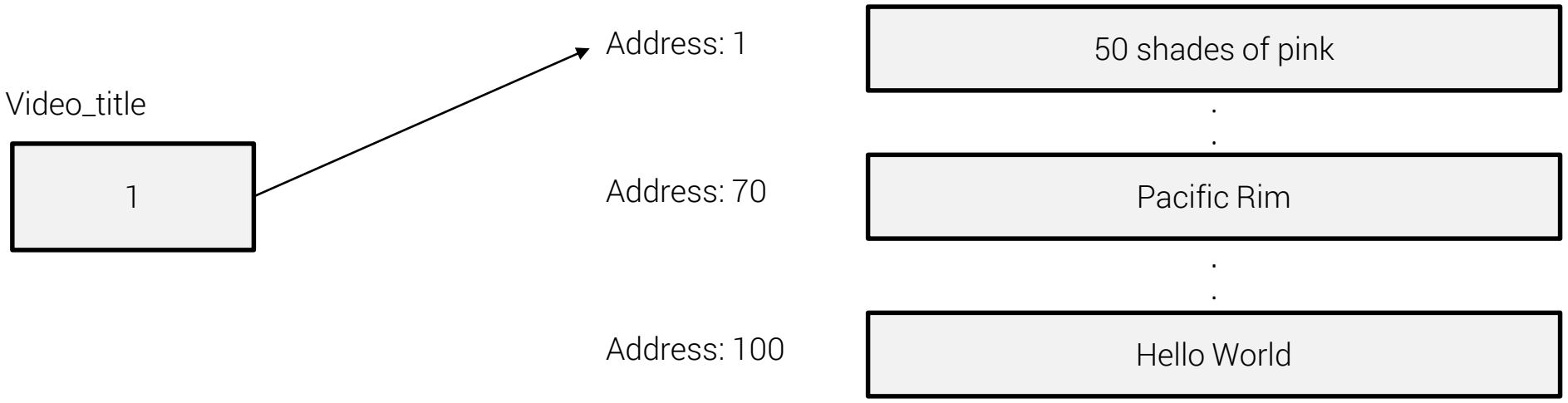
- String type variables are also known as reference variables. String variable stores the "address"/reference of the memory location where the text is stored.



HOW ARE REFERENCE VARIABLES STORED?

- String type variables are also known as reference variables. String variable stores the "address"/reference of the memory location where the text is stored.

```
video_title = ‘Pacific rim’
video_title = ’50 shades of pink’
```



IN-CLASS PRACTICE: YOUTUBE ENGAGEMENT RATE*

- One of the common metrics used to evaluate YouTube channel success is the engagement rate (formula given below). Let's try to declare the relevant variables and calculate the engagement rate for a YouTube channel.

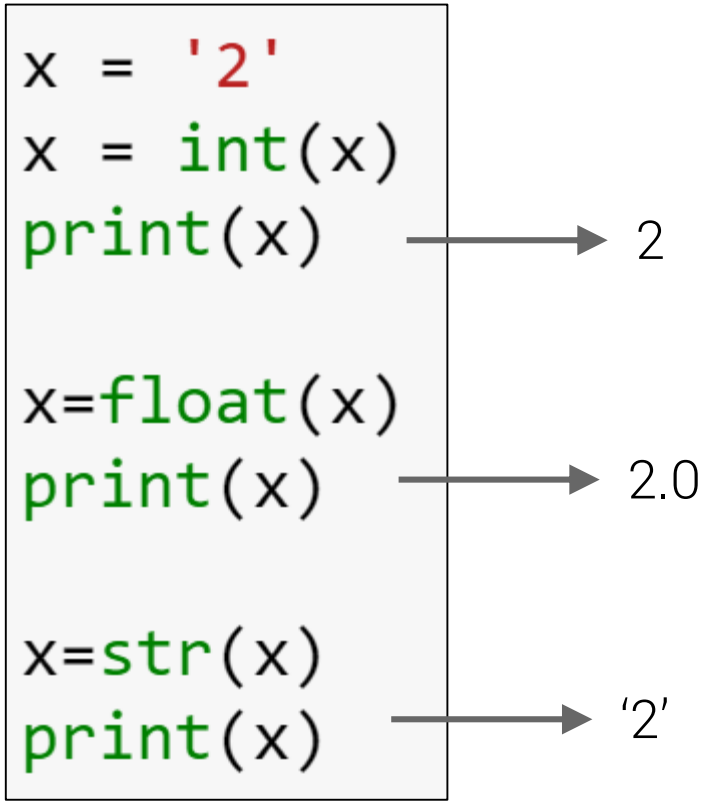
```
engagement_rate = video_comments/video_views
```

http://tubularinsights.com/youtube-analytics-viewer-engagement/

CONVERSION FUNCTION

- Data are usually dirty
- Attributes which are suppose to be Integer might be stored as String, etc.
- Parsing a variable type into another:
  - int(x) : Convert x into an integer
  - float(x): Convert x into a float
  - str(x): Convert x into a string

CONVERSION FUNCTION: CAVEAT



A variable of one type can be converted into another type

```
x='a'
x = int(x)
```

However, conversion can only be done if data can be converted.

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-2-3992a514ad90> in <module>()
      1 x='a'
----> 2 x = int(x)

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

STRING SPECIAL PROPERTIES

- String slicing
- String concatenation

STRING SLICING

- The most common data type we collect are text-based
- Refer to here for the full list of Python String functions: https://www.tutorialspoint.com/python/python_strings.htm
- Interesting function - String Slicing []:

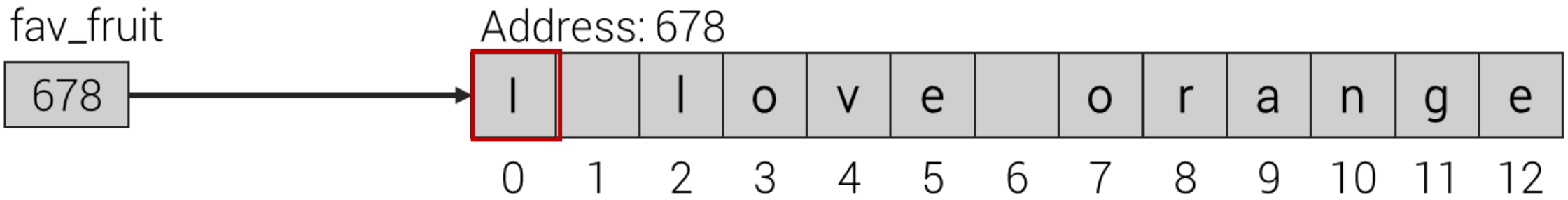
STRING SLICING

- Slicing a String:

```
fav_fruit = 'I love orange'
print(fav_fruit[0])
```

Output:
'I'

Structure of string data:



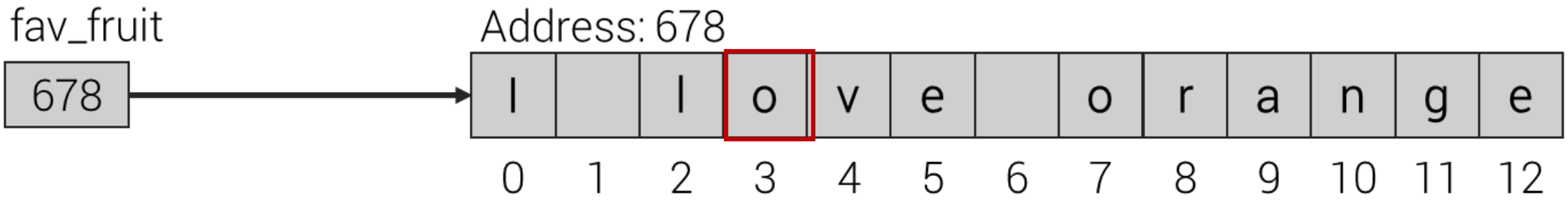
STRING SLICING

- Slicing a String:

```
fav_fruit = 'I love orange'
print(fav_fruit[3])
```

Output:
'o'

Structure of string data:



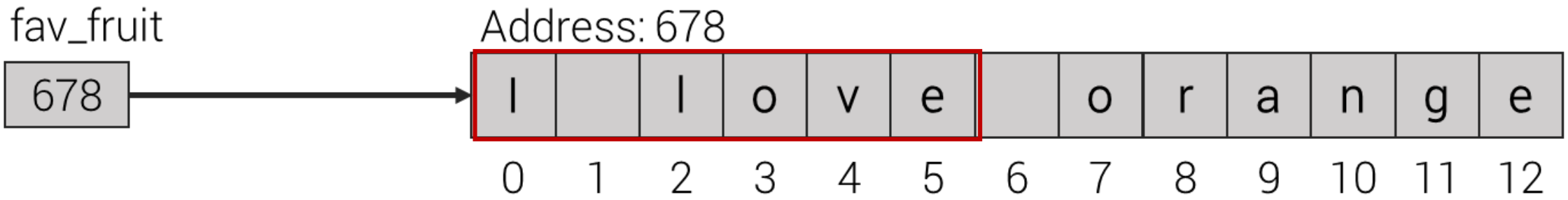
STRING SLICING

- Slicing a String:

```
fav_fruit = 'I love orange'
print(fav_fruit[0:6])
```

Output:
'I love'

Structure of string data:



IN-CLASS PRACTICE: STRING SLICING*

- Try these:
  - print(fav_fruit[7:])
  - print(fav_fruit[-1])
  - print(fav_fruit[-4:])
  - print(fav_fruit[:-4])
  - print(fav_fruit[::-1])

STRING CONCATENATION

- Another special property of string is that they can be combined, or rather concatenated (Official programming term)

```
Video_tit = ‘dota2FTW’
new_word = Video_tit + Video_tit
print(new_word )
```

Output: ‘dota2FTWdota2FTW’

IN-CLASS PRACTICE: STRING CONCATENATION*

- Try these:
  - print(video_tit + ‘hello world’)
  - print(video_tit + ‘video_tit’)
  - print(video_tit + video_tit[0:5])
  - print(video_tit + 5)
  - print(video_tit + str(5))

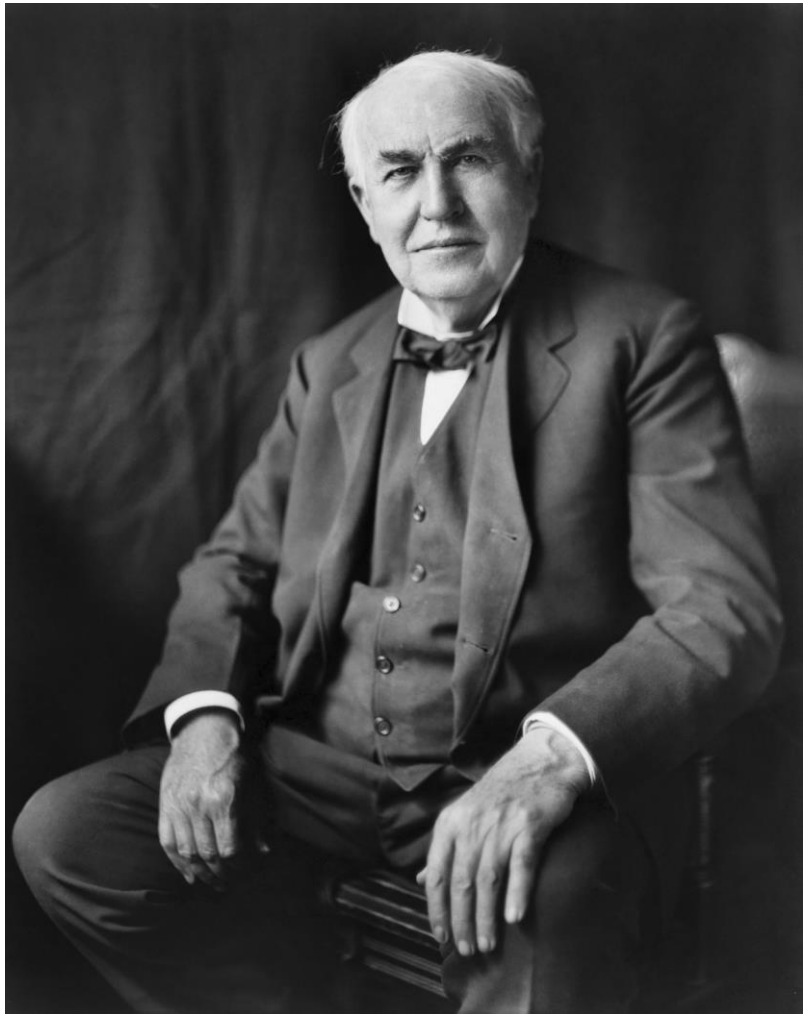
ART OF DEBUGGING

- Notion of debugging
- Type of bugs/errors
- Syntax error
- Runtime error
- Logic error
- Basic debugging techniques

NOTION OF DEBUGGING

- “It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise— this thing gives out and [it is] then that **'Bugs'** — as such little faults and difficulties are called—show themselves and months of intense watching, study and labour are requisite before commercial success or failure is certainly reached.”

- Thomas Edison, 1878



NOTION OF DEBUGGING

- De-bugging a program is the process of finding and resolving errors
- No programmers is able to write a complete program from scratch without any errors. In fact, even in complete programs such as Microsoft Word, there are often sporadic bugs here and there
- As such, other than writing codes, one other important skill a skilled programmer must possess is the ability to debug

TYPES OF BUGS/ERRORS

- **Syntax errors**: The code does not follow the rules of the language; for example, a single quote is used where a double quote is needed; a colon is missing; a keyword is used as a variable name.
- **Exceptions**: In this case, your code is fine but the program does not run as expected (it “crashes”). For example, if your program is meant to divide two numbers, but does not test for a zero divisor, a run-time error would occur when the program attempts to divide by zero.
- **Logic errors**: These can be the hardest to find. In this case, the program is correct from a syntax perspective; and it runs; but the result is unanticipated or outright wrong. For example, if your program prints “2+2 = 5” the answer is clearly wrong.

SYNTAX ERRORS

- print(“hello world’) ⟶ Syntax error (delimiters don’t match)
- prnt(“hello world”)

- **Syntax errors** occurs when the Python parser does not understand a line of code!

EXCEPTIONS

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
- Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs.
- Examples of common Exceptions:
  - ZeroDivisionError
  - NameError
  - TypeError

EXCEPTIONS: ZERODIVISIONERROR

- The following is an example of exception, where the code doesn't run even though there is nothing wrong in terms of syntax

```
a = 13 + 15
print(a/0)
```

LOGIC ERRORS

- The following is an example of exception, where the code is deemed to be wrong even though there is nothing wrong with the syntax and the program doesn't crash when executed.

```
video_views = 40
video_comments = 3
engagement_rate = video_views+video_comments
print(engagement_rate)
```

- We say a program has a logic error when the program does not do what a programmer expects it to do
- In the above, we have a certain expected output of the engagement_rate, and but the program will return a wrong engagement_rate

BASIC DEBUGGING TECHNIQUES

- **Rule 1:** Set small incremental goals for your program. Don't try and write large programs all at once. Stop and test your work often as you go by printing out the intermediate outputs. Celebrate small successes!
- **Rule 2:** Use comment to have Python ignore certain lines that are giving you trouble
- **Rule 3:** For large programs, use print statement to print out intermediate outputs in order to check the codes
- **Rule 4:** Understand what kind of error you are facing, and Google for potential solutions. Usually, stack overflow should have the answer you need!

Note: We will cover more about debugging different types of questions in future

IN-CLASS PRACTICE: DEBUGGING

- The following codes have some bugs. Using the techniques covered, resolve all of the bugs:

```
video_views = ‘300’
video_likes = 10

Engagement = video_likes / vide0_views
print(“The engagement rate is: " + Engagement)
```

PRINT & FORMAT FUNCTION (OPTIONAL)

- Line endings
- Separating print() function arguments
- Combining both line endings & separators
- Escape characters
- String concatenation
- String repetition
- Format function (String, numbers, percentage, integers)

LINE ENDINGS

- When using the `print()` function you probably have noticed that Python automatically places a newline character at the end of each line
- You can override this behaviour and have Python use a character of your choice by using the optional 'end' argument when using the `print()` function

Example:

```
print('one', end=' ')
print('one', end=' ')
>>one one
```

COMBINING BOTH LINE ENDINGS & SEPARATORS

- You can use both the ‘sep’ and the ‘end’ arguments at the same time in the same print() function call.

Example:

```
print(‘a’,‘b’,‘c’,sep=‘*’,end=‘ ’ )
>>a*b*c
```

ESCAPE CHARACTERS

- Most programming languages support an “escape character” that allows you to perform special actions inside the confines of a delimiter.
- In Python the escape character is the “\” character
- It causes Python to treat the next character as a “special” character – in most cases this means that you will ignore the next character and prevent it from interfering with your delimiter

Example:

```
print('Hi, I\'m Harry Potter, your professor’)
>> Hi, I'm Harry Potter, your professor
```

ESCAPE CHARACTERS

- There are a number of special characters you can use in conjunction with the escape character to perform special string operations.
- For example – "`\n`" – forces a line break.

Example:

```
print('line 1 \n line 2 \n line 3')
>>          line 1
      line 2
            line 3
```

ESCAPE CHARACTERS

- Example – “\t” – forces a tab:

```
x = 5
y = 4

print('X', '\t', 'Y', '\t', 'X*Y’)
print(x, '\t', y, '\t', x*y)

>>        X                    Y          X*Y
      5               4          20
```

FORMAT() FUNCTION

- The `format()` function can also be used to generate a printable string version of a float or integer number
- `format()` takes two arguments – a number and a formatting pattern (expressed as a string)
- `format()` returns a string which can be treated like any other string (i.e. you can print it out immediately, store its value in a variable, etc)

FORMAT() FUNCTION: LIMITING DECIMAL PRECISION

```
a = 1/6
print(a)
>> 0.16666666666666666666

b =format(a, '.2f')
print(b)
>> 0.17
```

FORMAT() FUNCTION: FORMATTING PATTERNS

```
a = 10000/6

b =format(a, '.2f') #format a as a 2 digit float

c =format(a, '.5f') #format a as a 5 digit float

d =format(a, ',.5f') #5 digit float + comma separators
```

FORMAT() FUNCTION: FORMATTING PERCENTAGES

```
a = 0.52

print(format(a, '%'))
>> 52.000000%

print(format(a, '.2%'))
>> 52.00%

print(format(a, '.0%'))
>> 52%
```

SUMMARY

- Rules of variables
- Data types & user input
- Primitive vs reference variables
- String slicing
- String concatenation
- Format function (String, numbers, percentage, integers)
- syntax errors, exceptions and logic errors
- Techniques of debugging