2. Introduction to Python

DataKirk

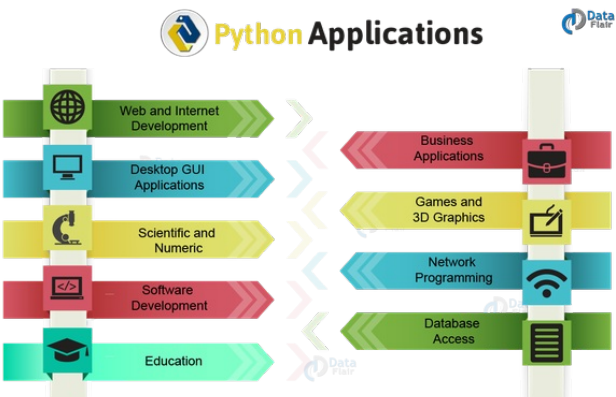
Talk Overview

1. What is Python?
2. How to run Python code
3. Python Basics

What is Python?

What is Python?

Python is a general purpose, high-level programming language



DataKirk

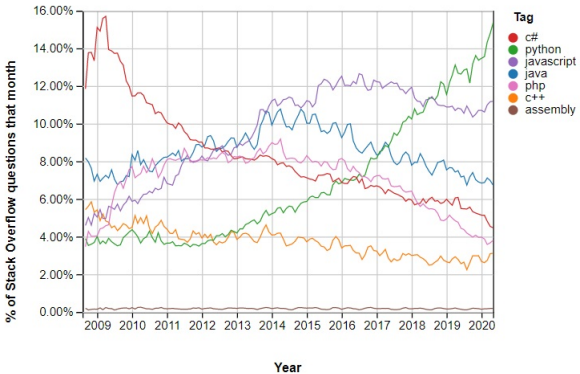
What is Python?

It's is popular, and growing fast

Aug 2019	Aug 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.028%	-0.85%
2	2		C	15.154%	+0.19%
3	4	^	Python	10.020%	+3.03%
4	3	v	C++	6.057%	-1.41%
5	6	^	C#	3.842%	+0.30%
6	5	v	Visual Basic .NET	3.695%	-1.07%
7	8	^	JavaScript	2.258%	-0.15%
8	7	v	PHP	2.075%	-0.85%
9	14	^^	Objective-C	1.690%	+0.33%
10	9	v	SQL	1.625%	-0.69%

What is Python?

It's is popular, and growing fast



What is Python?

Programmers love it!

Example application: Instagram

Instagram

How does it work?

Step 1: write your Python code

```
In[1]: x = 2
       y = 3
       print(x + y)
```

Step 2: run the code

Step 3: get the output

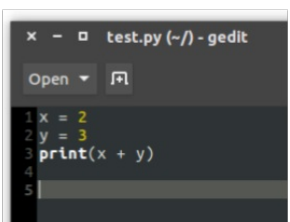
```
Out[1]: 5
```

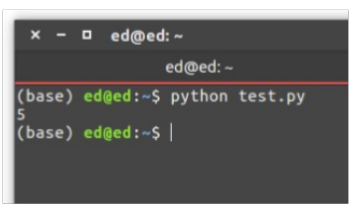
How does it work?

Option 1: Write code in a text editor, then run from the command prompt

- A text editor is a place to read and write code. Examples include Notepad++ and atom but not MS word.
- The 'command prompt/line' or 'terminal' is a place where you can run code . The look a little different on PC/Mac/Linux but are all essentially the same.

```
C:\Users\Me\path\to\file> python your_file_here.py
```

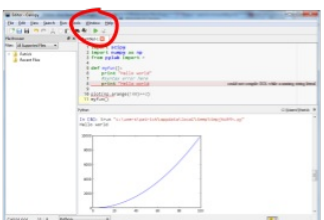


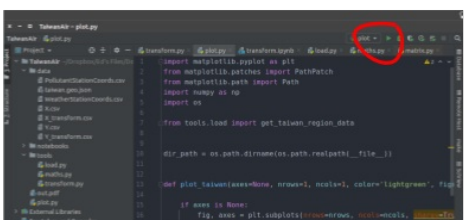


How does it work?

Option 2: Use an IDE that has 'run' button built in

- Some Interactive Development Environments (IDEs) (e.g. Pycharm) have a click-to-run button.





How does it work?

Option 3: run in interactive mode from the command prompt

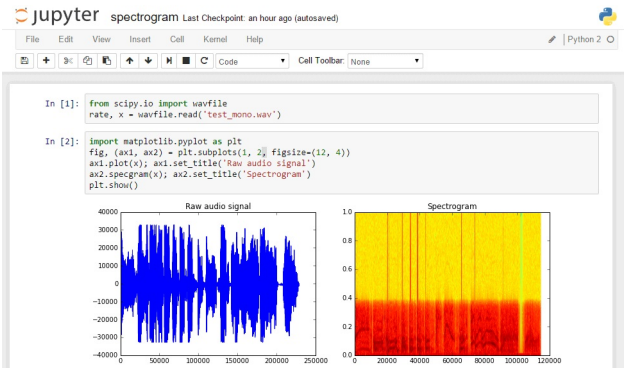
- To start an interactive shell, open the terminal app or command prompt and type `python` to enter.

```
> python
Python 3.7.5 (default, Nov 20 2019, 09:21:52)
[GCC 9.2.1 20191008] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- By default it shows `>>>`, where you can type Python code. You can type any expression or statement here.

How does it work?

Option 4: Use Jupyter notebooks (this is what we'll be doing)



Jupyter notebooks

- Code is written inside 'cells' which can be executed individually.

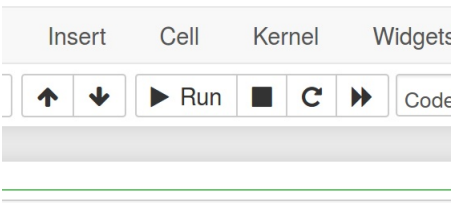
```
In [1]: print('Hello world')
```

```
Out[1]: 'Hello world'
```

- Nice interface where you can see plots and examine data.
- You can also create 'markdown' cells. These allow for normal text, bullet points, tables, maths equations and more.

Jupyter notebooks

- To run a cell, you can either click the 'run' button at the top bar



- Or use the shortcuts `ctrl + enter` or `shift + enter`.

The print function

- The print function is used to display data on the screen

```
In [2]: print('Hello world')
```

```
Out[2]: 'Hello world'
```

- It has nothing to do with actual printers! You can just think of 'print' as really meaning 'display'.

```
In [3]: print(5)
```

```
Out[3]: 5
```

The print function

- Each time you use print, the output will appear on a new line.

```
In[4]:  print(10)
        print(20)
```

```
Out[4]: 10
        20
```

- You can also print multiple things on one line by giving `print` each item separated by a comma.

```
In[5]:  print(1, 2, 3, 4)
```

```
Out[5]: 1, 2, 3, 4
```

Python as a calculator

- One use of Python is just as a regular calculator.

```
In [6]: print(2 + 2)
```

```
Out[6]: 4
```

Operation	Maths symbol	Python symbol	Example
Addition	$+$	`+`	`2 + 2`
Subtraction	$-$	`-`	`5 - 2`
Multiplication	$\times$	`*`	`2 * 2`
Division	$\div$	`/`	`6 / 2`
Powers	x^y	`**`	`5 ** 2`

Maths examples

```
In[7]:  print(0.99 * 100)
        print(8 / 2)
        print(5 ** 2)
```

```
Out[7]: 99
        4
        25
```

- The order of operations follows BODMAS (Brackets, Orders, Division, Multiplication, Addition, Subtraction).

```
In[8]:  print(10 + 10 / 2)
        print((10 + 10) / 2)
```

```
Out[8]: 15
        10
```

Error messages

- Error messages occur when Python is unable to compute something.

```
In [4]: print(1 / 0)
```

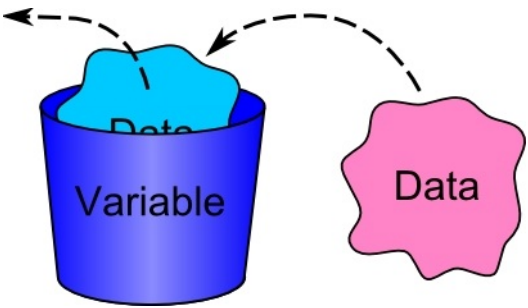
```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-4-3ec96714f820> in <module>
----> 1 print(1 / 0)

ZeroDivisionError: division by zero
```

- When you get an error message you will see:
  1. The line in your code that caused the error.
  2. A short (possibly quite cryptic) description of the error.

- A good strategy if you don't understand the error is to google it. Often you will find forums and discussion pages such as stackoverflow where people have encountered and solved similar issues.

Variables

- Variables are like a box, with a name, that stores some data.
- This data can be read and modified.



- You can always check at any time what's inside the box.

Variables

- In Python the `=` symbol assigns the data on the right to the name on the left.

```
In[9]:  pi = 3.14159
        radius = 3
```

- When you `print` a variable, you see what's inside it

```
In[10]:  print(pi)
         print(radius)
```

```
Out[10]: 3.14159
         3
```

Variables

- Variables can then be used in calculations later

```
In[11]:  print(2 * pi * radius)
         print(pi * radius ** 2)
```

```
Out[11]: 18.84954, 28.27431
```

- Or used to make new variables

```
In[11]:  circumference = 2 * pi * radius
         area = pi * radius ** 2
         print(circumference, area)
```

```
Out[11]: 18.84954, 28.27431
```

Variables

- There are some rules about what names you can use for variables.

    1. Variable names can only contain letters, digits, and underscores (`_`) and cannot start with a digit.
    2. Variables cannot contain spaces in their name.
    3. Variables cannot contain any special characters such as `", $, %, &` etc.
    4. Variables are case sensitive, so `my_variable` is not the same as `My_Variable`.

Variables

- Below are some examples of valid variable names, and some invalid variable names.

Valid name	Invalid name	Reason invalid
`a2`	`2a`	Starts with a number
`my_variable`	`my variable`	Contains a space
`a_long_variable`	`a long variable`	Contains multiple spaces
`Two_Pounds`	`£2`	Contains a special character
`a_plus_b`	`a+b`	Contains a special character

Variables

- Variables can be overwritten by just reassigning them.

```
In[12]:  my_number = 5
         print(my_number)
         my_number = 6
         print(my_number)
```

```
Out[12]: 5
         6
```

- When reassigning a variable, you can use the variable's current value.

```
In[13]:  my_number = my_number + 1
         print(my_number)
```

```
Out[13]: 7
```

More on numbers

- In python, and many other languages, numbers come in two distinct types: `int`s (whole numbers) and `float`s (numbers with a decimal point).
- `int` is short-hand for "integer".
- `float` is short-hand for "floating point number".
- When it comes to doing maths with these numbers, there's not much difference. However, it's good to be aware.
- You can use the keyword `type` to find out they type

```
In [14]: my_int = 5
         my_float = 11.5
         print(type(my_int))
         print(type(my_float))
```

```
Out[14]: int
         float
```

Text data

- In addition to numbers, Python also handles text. Any piece of text needs to be enclosed in either single `'` or double `"` quotes.

```
In[15]:  print('a piece of text')
```

```
Out[15]: 'a piece of text'
```

- Like numbers, text can be assigned to a variable.

```
In[16]:  my_text = "another piece of text :)"
         print(my_text)
```

```
Out[16]: "another piece of text :)"
```

Text data (strings)

- Long pieces of text that flow over multiple lines need to be enclosed in either triple single quotes `'''` or triple double quotes `"""`.

```
In[17]: long_text = '''This is a long piece of text
    that flows over multiple lines'''
```

- In python we refer to pieces of text data as "strings". We can use the python `type` function to check the type of text data.

```
In[18]:  my_string = 'Hello, nice to meet you'
         print(type(my_string))
```

```
Out[18]: str
```

Working with strings

- Strings can be added together ("concatenated")

```
In [19]: first_name = 'Daveed'
         second_name = 'Diggs'
         full_name = first_name + ' ' + second_name
         print(full_name)
```

```
Out[19]: 'Daveed Diggs'
```

- Strings can also be multiplied by a number

```
In [20]: print('=' * 10)
```

```
Out[20]: '=========='
```

Strings

- However, strings cannot be subtracted

```
In [21]: print('a' - 'b')
```

```
TypeError                       Traceback (most recent call last)
----> 1 print('a' - 'b')
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

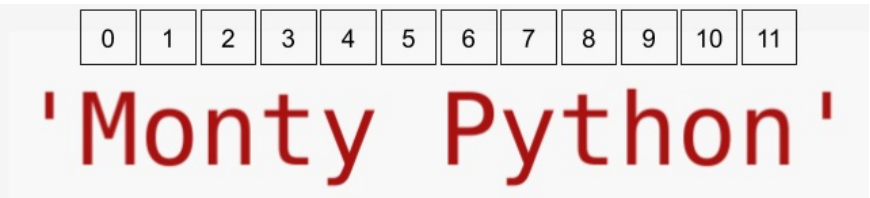
- And you can't multiply a string by a string

```
In[21]: print('hello' * 'h')
```

```
TypeError                       Traceback (most recent call last)
----> 1 print('hello' - 'h')
TypeError: cant multiply sequence by non-int of type 'str'
```

String-indexing

- A string is made up of individual characters. (That's where it gets the name - its a *string* of characters).
- Characters in a string are numbered starting from zero.



String-indexing

- Individual characters can be accessed using square brackets. This is called indexing.

```
In[22]:  my_string = 'Monty Python'
         print(my_string[6])
```

```
Out[22]: 'P'
```

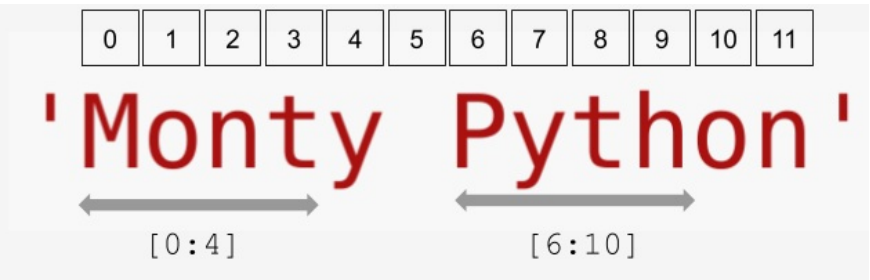
- Always remember that that characters start from zero.

```
In[22]:  print(my_string[0])
```

```
Out[22]: 'M'
```

String-indexing

- Sub strings (a chain of characters within a string) can be accessed using square brackets as `[start:stop]`. Here, `start` is inclusive, `stop` is non-inclusive.

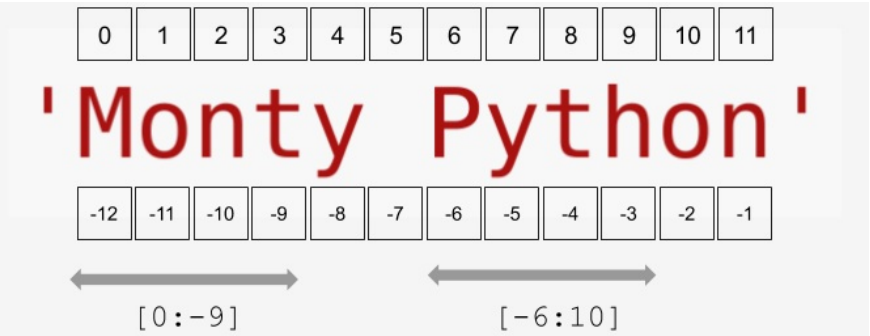



```
In [17]: print(my_string[6:10])
```

```
Out[17]: 'Pyth'
```

String-indexing

- You can also refer to individual characters from the end of the string, using negative numbers.



Lists

- A list is data type that holds an ordered sequence of other pieces of data.
- Lists can hold anything: numbers, strings, even other lists.
- A list is created by enclosing a set of items between square brackets, separated by commas.
- Each item in the list can be of a different type.

```
In [18]: my_list = [42, 'dog', -0.5, 'cat', 'fish']
         print(my_list)
```

```
Out[18]: [42, 'dog', -0.5, 'cat', 'fish']
```

Lists

- You can find out the length of a list by using the `len()` special function.

```
In [21]: print(len(my_list))
```

```
out[21]: 5
```

- Lists are their own distinct data type

```
In [21]: print(type(my_list))
```

```
Out[21]: list
```

Lists

- Just as with strings, individual items or subsections can be accessed by indexing.

```
In [22]: to_do = ['feed cat', 'email Dave', 'run', 'pick up kids']
         print(my_list[0])
         print(my_list[2])
         print(my_list[1:3])
```

```
Out[22]: 'feed cat'
         'run'
         ['email Dave', 'run']
```

Lists

- As well as retrieving, you can also override individual items in a list.

```
In [23]: to_do[0] = 'feed dog'
         print(to_do)
```

```
Out[23]: ['feed dog', 'email Dave', 'run', 'pick up kids']
```

- Or multiple items.

```
In [23]: to_do[1:3] = ['email Enayi', 'jog']
         print(to_do)
```

```
Out[23]: ['feed dog', 'email Enayi', 'jog', 'pick up kids']
```

Lists

- New items can be added to the end of a list using the `.append()` command.

```
In [24]: to_do.append('read')
         print(to_do)
```

```
Out[24]: ['feed dog', 'email Enayi', 'jog', 'pick up kids', 'read']
```

- It is common to start with an empty list, and then add items one by one.

```
In [25]: empty_list = []
         empty_list.append('shoe')
         empty_list.append('glove')
         empty_list.append('sock')
         print(empty_list)
```

```
Out[25]: ['shoe', 'glove', 'sock']
```

Thanks!