Introduction to Python

by Reynaldo Morillo

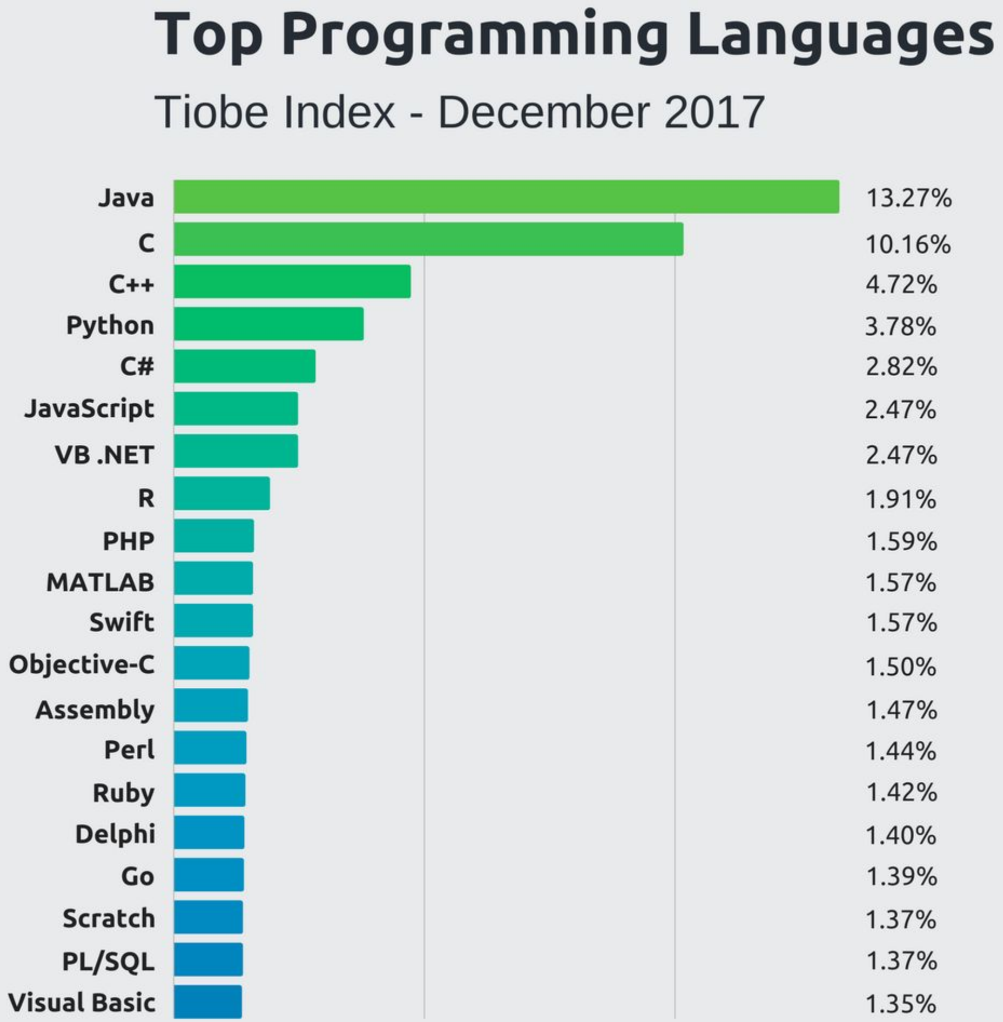
Contents

Tips

- Dark blue words are links (except these words)
- While I'm presenting live, you most likely will not be able to keep up with what I'm showing. So you might be better off listening and asking questions, than trying it yourself on the spot. However, if you think you can, feel free to do so.
- I'm only going to show you enough to get started with just about anything. Everything here can be explored in excruciating detail, but I will not do that. Too much to cover, too little time.
- However, I encourage exploring these topics in more detail because you'll be able to manipulate the language better, and it will help you solve seemingly weird bugs

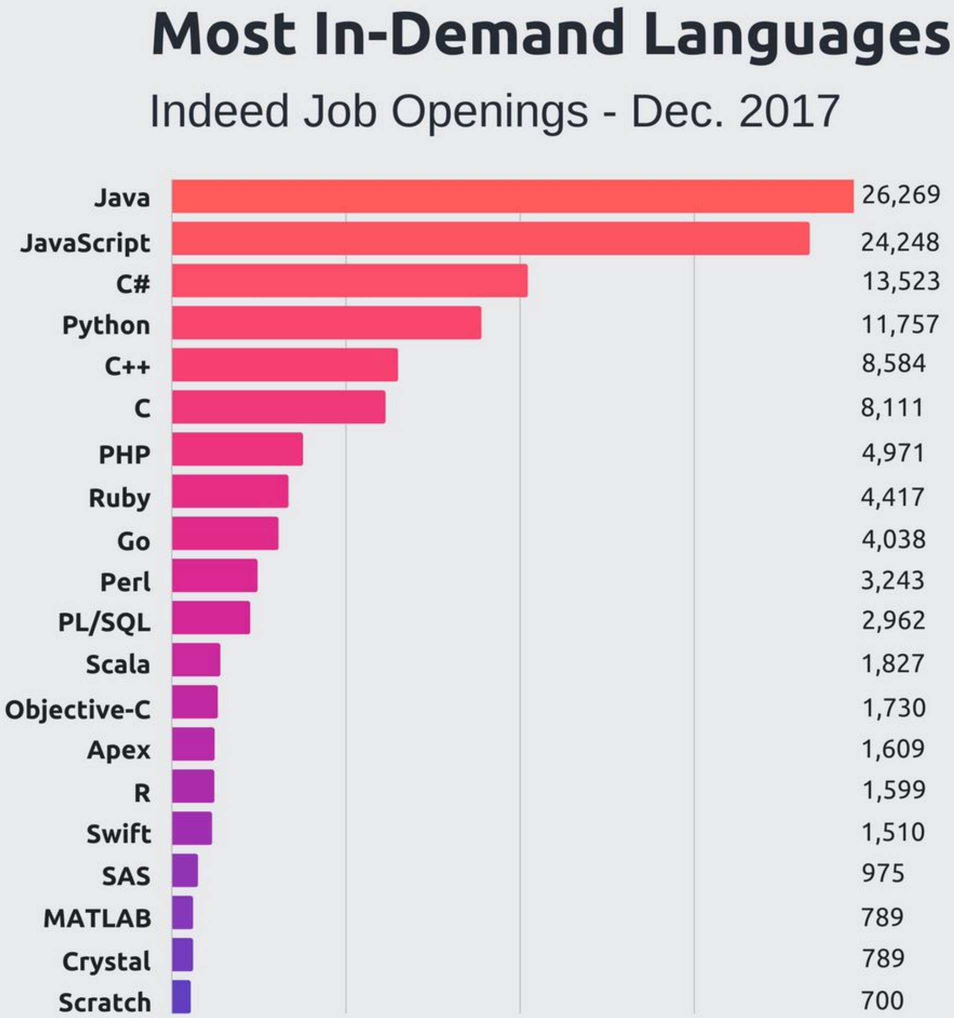
Why Learn Python?

Python is on the rise!



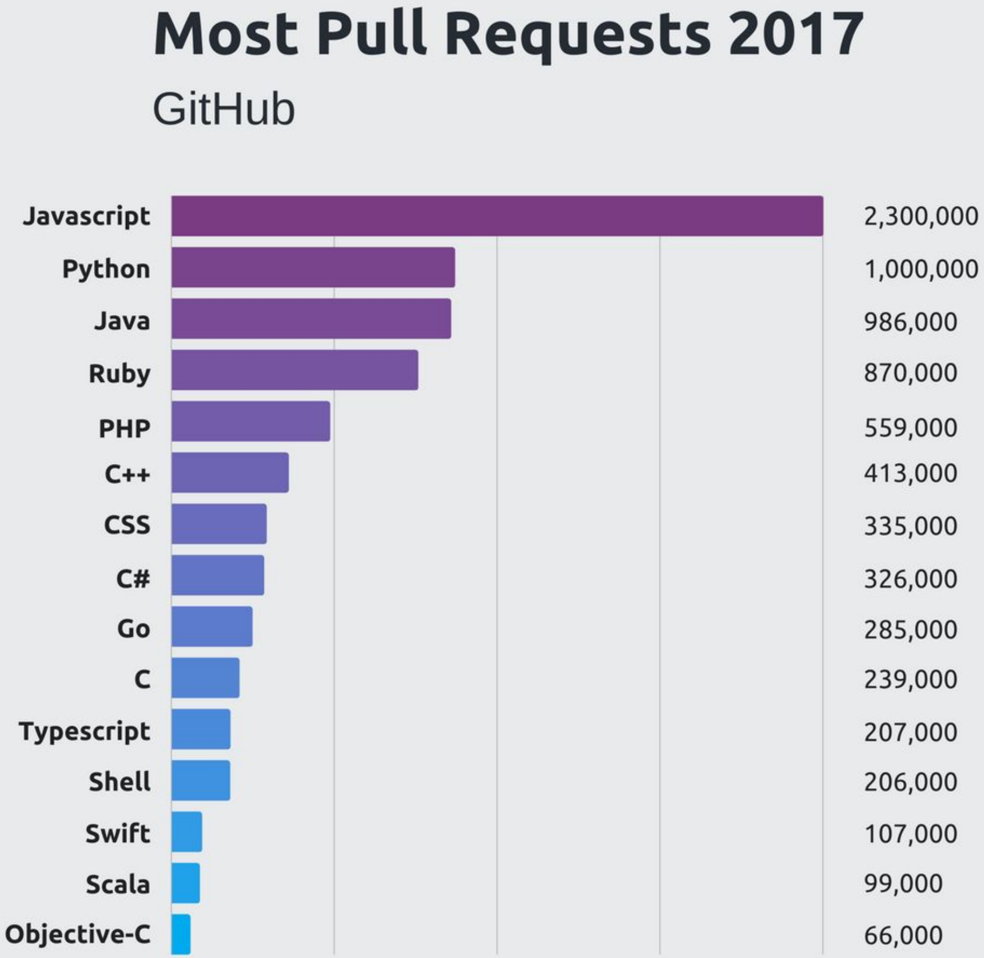
Credits to Ben Putano December 18, 2017

Python is on the rise!



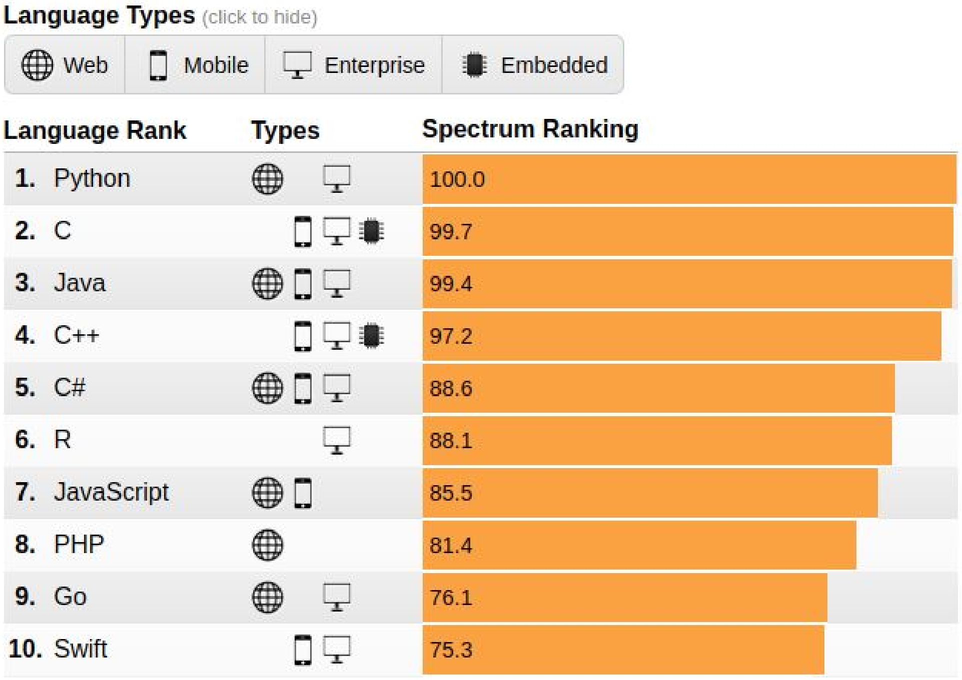
Credits to Ben Putano December 18, 2017

Python is on the rise!



Credits to Ben Putano December 18, 2017

Python is on the rise!



The 2017 Top Programming Languages

Credits to Nick Diakopoulos and Stephen Cass

Where is Python being used?

- Artificial Intelligence
- Big Data Analytics
- Robotics

About Python

Language Features

- General Purpose Programming
- Interpreted
- Dynamically Typed
- Automatic Memory Management

Programming Paradigms

- Object Oriented Programming
- Procedural
- Functional
- Imperative

History

First appeared on February 20th, 1991

Created by Guido van Rossum

Jan 1994 — **Python 1.0**

First major release

Oct 2008 — **Python 2.6**

Popular release until 2.7

Dec 2008 — **Python 3.0**

Breaks backwards compatibility, but makes the language more consistent in design, and more efficient for data analytics.

Jul 2010 — **Python 2.7**

A big favorite at the moment, where there is a huge code base of modules that are compatible with this version, and retains backwards compatibility.

Dec 2016 — **Python 3.6**

Latest stable release that machine learning is jumping on. Moving forward machine learning has favored 3.5 and after because it adds some conveniences for data manipulation.

Credits to Wikipedia

Setup

There are many flavors of Python

- Cpython (A.K.A. Python)
- Cython
- Jython
- Iron Python
- Pypy
- Anaconda Python

Why Anaconda Python?

Pros

- It is not different from CPython (i.e. the original Python)
- It’s essentially just a Package Manager and Environment Manager
- Developed with Data Science in mind
    - Makes Installing packages easy for everyone, regardless of Operating System
    - A collection of over 1,000+ open source packages (mostly data science specific)

Cons

- There are packages that are not available in Anaconda
- Might add a little complexity to a project setup, but it’s something you’ll get used to
- Anaconda is huge like (3 Gigabytes), but you can opt to use Miniconda instead

Anaconda or Miniconda?

Anaconda is so large because it comes with essentially all the data science packages you need, and then some.

Miniconda only comes with the essentials, which is essentially Python.

You can pick whichever you like, because they work the same way. Any instructions for one, works for other. It's just that for Miniconda you will have to select which packages you want to download and install.

In this presentation I'm going to use Miniconda 3.6 (64-bit), which comes with Python 3.6. I will install additional libraries later.

What if I already have Python?

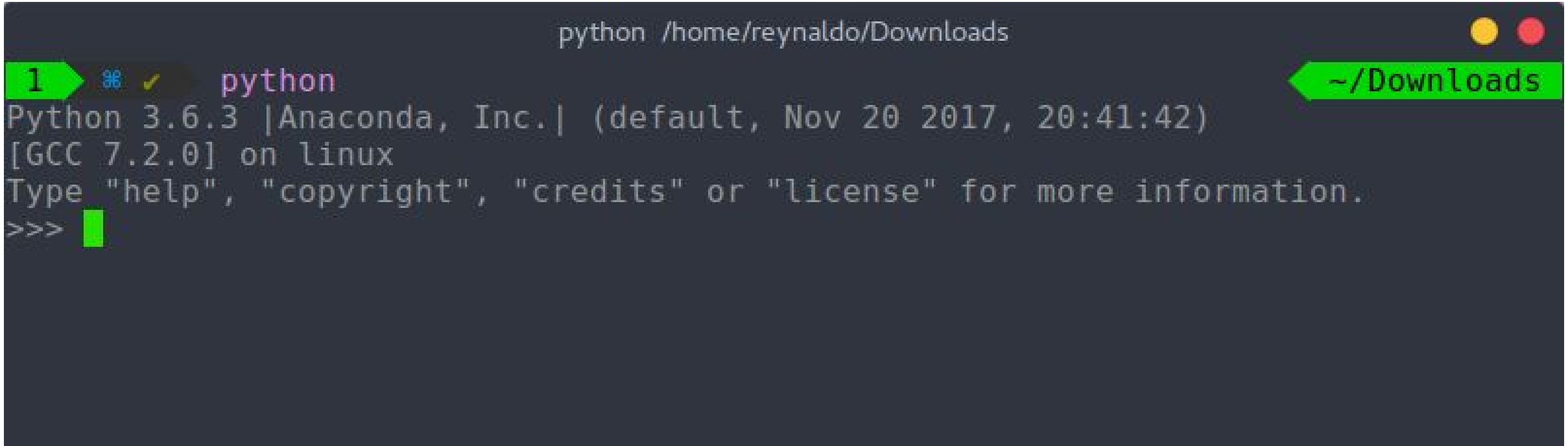
Not a problem!

Anaconda and Miniconda operate on the project level, not the system level, so it shouldn't bother your current python installation.

It might remap the python command to Anaconda's python, but functionally they're the same. If that bothers you, you can remap it back to the original Python. It might just be a matter of adjusting your PATH environment variable.

Now what?

At this point you should have Python (any python flavor) installed and your ready to dive into the Syntax and The Basics



Syntax

Code Example

```
age = 1  # No need to declare its type (integer)
length = 12.1  # automatically recognized as a float
species = 'python'  # string using single quotes

# Check if the species is abnormally large
# Right now we only can check pythons ...
if species == 'python' and age < 4 and length > 10:
    print("That python is huge!!!" )
    estimated_weight = age * length * 13.4 + 2.2  # I made this up
    print("It must weigh " + str(estimated_weight) + " lbs")
```

Enough to understand example

Comments are followed by #, which are non-executable text in the code, usually used for notes

You can save **values** into **variables** using = (i.e. assignment operator)

The variables here are: *age*, *length*, *species*, and *estimated_weight*

Their corresponding values are on the opposite side of the =

The type of these variables are automatically inferred

```
age = 1  # No need to declare its type (integer)
length = 12.1  # automatically recognized as a float
species = 'python'  # string using single quotes

# Check if the species is abnormally large
# Right now we only can check pythons ...
if species == 'python' and age < 4 and length > 10:
    print("That python is huge!!!" )
    # I made this up
    estimated_weight = age * length * 13.4 + 2.2
    print("It must weigh " + str(estimated_weight) + " lbs")
```

Enough to understand example

Strings are text values created by surrounding text in “ ” (double quotes) or ‘ ’ (single quotes).

A conditional statement controlling the flow of the program execution is represented with the if statement.

In this case we go into the indented code block if the *species* is a python and the *age* of the python is less than 4, and the *length* of the python is greater than 10 (i.e. our acceptance condition). Otherwise, the program ends immediately after seeing the condition fail.

```
age = 1  # No need to declare its type (integer)
length = 12.1  # automatically recognized as a float
species = 'python'  # string using single quotes

# Check if the species is abnormally large
# Right now we only can check pythons ...
if species == 'python' and age < 4 and length > 10:
    print("That python is huge!!!" )
    # I made this up
    estimated_weight = age * length * 13.4 + 2.2
    print("It must weigh " + str(estimated_weight) + " lbs")
```

Enough to understand example

Inside the indented code block we're invoking a function called **print**, that outputs text to the screen/terminal/console/whatever ...

Note that print is using parentheses to print the string it's given. That's a noticeable difference from Python 2.7.

You can perform mathematical operations on variables. As seen to calculate the *estimated_weight*.

```python
age = 1  # No need to declare its type (integer)
length = 12.1  # automatically recognized as a float
species = 'python'  # string using single quotes

# Check if the species is abnormally large
# Right now we only can check pythons ...
if species == 'python' and age < 4 and length > 10:
    print("That python is huge!!!" )
    # I made this up
    estimated_weight = age * length * 13.4 + 2.2
    print("It must weigh " + str(estimated_weight) + " lbs")
```

Enough to understand example

You can combine multiple strings together to form a single string. This is called string concatenation.

Note that *estimated_weight* is not a string (it's a float). So you must convert it into a string using **str** to combine it with the other strings.

What's the output?

```
age = 1  # No need to declare its type (integer)
length = 12.1  # automatically recognized as a float
species = 'python'  # string using single quotes

# Check if the species is abnormally large
# Right now we only can check pythons ...
if species == 'python' and age < 4 and length > 10:
    print("That python is huge!!!" )
    # I made this up
    estimated_weight = age * length * 13.4 + 2.2
    print("It must weigh " + str(estimated_weight) + " lbs")
```

Enough to understand example

That python is huge!!!
It must weigh 164.33999999999997 lbs

```
age = 1  # No need to declare its type (integer)
length = 12.1  # automatically recognized as a float
species = 'python'  # string using single quotes

# Check if the species is abnormally large
# Right now we only can check pythons ...
if species == 'python' and age < 4 and length > 10:
    print("That python is huge!!!" )
    # I made this up
    estimated_weight = age * length * 13.4 + 2.2
    print("It must weigh " + str(estimated_weight) + " lbs")
```

A Twist (Boilerplate)

In terms of output compared to the prior program, it has no difference.

Why do this?

At the end of the script you can see the conditional statement checking if the **__name__** of the program is **__main__**.

This makes sure you're executing this program directly rather than using it via an import

It will make sense later...

```
def main():
    age = 1  # No need to declare its type (integer)
    length = 12.1  # automatically recognized as a float
    species = 'python'  # string using single quotes

    # Check if the species is abnormally large
    # Right now we only can check pythons ...
    if species == 'python' and age < 4 and length > 10:
        print("That python is huge!!!" )
        # I made this up
        estimated_weight = age * length * 13.4 + 2.2
        print("It must weigh " + str(estimated_weight) + "
lbs")

if __name__ == '__main__':
    main()
```

The Basics Brace Yourself!

In Python almost everything is an object

Words of wisdom

What are Objects (Simply)?

It’s a **thing** that can **perform actions**

It can hold **information**. You can say, “**it knows**” certain **things**.

In a nutshell that’s about it, but I will give you more details later. This is enough to get by for now.

Assignment

Using the assignment operator = you can save values into variables.

What actually happens is your saving a **reference** to the value (which is actually an object). It seems like a small detail, but it actually makes a huge difference. It will be made clear later

Naming Rules

Variables can use alphanumeric characters and _ to create names. Variable names cannot start with a number.

Variable names should be descriptive of the values they refer to, even if it seems a little long. Trust me, you know it's okay if you ever program for MacOS / iOS.

Essentially use snake_case.

```
>>> estimated_weight = 205.3
>>> new_filename = 'new_file.txt'
>>> is_awake = True
>>> age = 5
>>>
```

Ignore the >>> it's from the python interpreter

Basic Data Types

- **Ints**: whole numbers, which can be as big as your computer memory can handle! In Python 3, there is not notion of **long** type. Only int.
- **Floats**: floats are any real number, it can be done in many different ways.
- **Strings**: Text, as long as your memory can handle. It is indexable.

```
>>> an_int = 5
>>> a_float = 5.23
>>> another_float = .1
>>> a_string = "foo"
>>> another_string = 'bar'
>>> one_more_string = "I said, 'LOL!'."
>>> one_more_string[0]  # Get first character
'I'
>>> one_more_string[-1]  # Get last character
'.'
>>> one_more_string[9:13]  # slice
'LOL!'
>>> len(one_more_string) # length of string
15
```

Data Type Conversion

For any of the basic data types you can get you can get a conversion between one and the other.

Except a string that looks like a float to an int.

```
>>> taco_price = "5.85"
>>> as_float = float(taco_price)
>>> as_float
5.85
>>> as_int = int(as_float)
>>> as_int
5
>>> new_taco_price = str(as_int)
>>> new_taco_price
'5'
>>> as_int = int(taco_price)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with
base 10: '5.85'
```

Indentation for Scoping

Scope: it's the environment in which variables exist, and defines its accessibility.

The Global Scope is the scope in which is accessible anywhere in the program, but things can get a little weird for functions.

Indentation declares the beginning and the end of a scope.

Scopes are nested inside each other by indentation. Hence why the local scope of the function is inside the global scope.

```python
# Global Scope
global_var = 'foo'

if global_var == 'foo':
    # This is still the global scope
    global_var2 = 'noodle'

def ex1():
    # The function's scope
    local_var = 'bar'
    print(global_var)
    print(global_var2)
    print(local_var)

ex1()
print(global_var)
print(global_var2)
print(local_var)  # this gives an error
```

Credits to Eliot

Indentation for Scoping

A scope that is nested inside another scope has access to variables in the scope above it (i.e. the scope that encapsulates it).

Hence why you can access the *global_var* from the **ex1** function scope.

It does not work the other way around. That is the encapsulating scope doesn’t have access to the scopes it encapsulates.

Hence why you cannot access *local_var*, which was defined in the local scope of the **ex1** function.

```python
# Global Scope
global_var = 'foo'

if global_var == 'foo':
    # This is still the global scope
    global_var2 = 'noodle'

def ex1():
    # The function's scope
    local_var = 'bar'
    print(global_var)
    print(global_var2)
    print(local_var)

ex1()
print(global_var)
print(global_var2)
print(local_var)  # this gives an error
```

Credits to Eliot

Indentation for Scoping

The indentation often used is whatever is mapped to the **tab** key. Depending on you IDE / Text Editor, that can be tab or 2 - 4 spaces. It’s arbitrary, but it’s recommend that you use either 2 or 4 spaces, because that’s the most common case.

Most of the time, your text editor will take care of the details, especially IDEs.

```python
# Global Scope
global_var = 'foo'

if global_var == 'foo':
    # This is still the global scope
    global_var2 = 'noodle'

def ex1():
    # The function's scope
    local_var = 'bar'
    print(global_var)
    print(global_var2)
    print(local_var)

ex1()
print(global_var)
print(global_var2)
print(local_var)  # this gives an error
```

Credits to Eliot

Data Structures

Data Structures

I'm just going to show you the essential ones. There are much more Data Types that you can make use of. The additional data types offer some performance features, and conveniences.

- Tuples
- Lists
- Dictionaries
- Sets

Lists

It’s array like data structure that can hold any combination of objects.

It’s indexable and has many of its own functions.

It’s very flexible, and used a lot in Python.

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple',
'banana']
>>> fruits[1]  # second element
'apple'
>>> fruits[0]  # first element
'orange'
>>> fruits[-1]  # last element
'banana'
>>> fruits[2] = 'pineapple'
>>> fruits
['orange', 'apple', 'pineapple', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange',
'grape']
```

Credits to Python Documentation

Tuples

Tuples are like lists, except that they re immutable, which essentially means they cannot be modified once they are created.

Strings and lists and tuples are part of the Sequence types, which share similar functions and attributes.

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Credits to Python Documentation

Sets

Sets are just like you think you are. An object that can hold a unique set of objects, with the standard set like operations like Union and Intersection.

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Credits to Python Documentation

Dictionaries

It computer science it's originally recognized as a Hash Table/Map.

It stores **key**, **value** pairs. Where a key can be any immutable object, and the value can be any object.

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Credits to Python Documentation

Conversion Functions

The conversion functions work between the sequence types, like lists, sets, tuples, and even strings.

```
>>> fruits
['orange', 'apple', 'pineapple', 'banana', 'kiwi',
'apple', 'banana']
>>> type(fruits)
<class 'list'>
>>> fruit_set = set(fruits)
>>> fruit_set
{'grape', 'banana', 'kiwi', 'apple', 'pear', 'orange'}
>>> type(fruit_set)
<class 'set'>
>>> fruit_tuple = tuple(fruit_set)
>>> fruit_tuple
('grape', 'banana', 'kiwi', 'apple', 'pear', 'orange')
>>> type(fruit_tuple)
<class 'tuple'>
>>> set('Hello World')
{' ', 'e', 'o', 'r', 'W', 'd', 'l', 'H'}
```

Operators

Arithmetic

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11
* Multiplication	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Comparison

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a!= b) is true.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Credits to Tutorials Point

Essential Operators

Operator (Magic)

Objects in python can have special actions associate with operators. As you already have seen with the **set** data structure object.

In Python you can create your own class objects with their own special actions associate with the operators. They're called magic methods.

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters
in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Control Flow

Control Flow

You can control the execution of the program using **if**, **elif**, and **else** statements.

You can chain a number of conditions together in a latter for exclusive execution.

Note after each condition specification there's a : (colon) which specifies the end of the condition, and implies that the next line there is an indented code block.

```
# My patented "Can I play tennis" machine
# I assume sunny whether with a slight breeze, and the finest air quality
temp_sensor_reading = 35
if temp_sensor_reading > 85:
    print("It's too hot outside!")
elif temp_sensor_reading == 85:
    print("Perfect!")
elif temp_sensor_reading < 85 and temp_sensor_reading > 45:
    print("Okay(ish)")
else:
    print("No tennis today :(")
```

Loops

For Loops

For loops are used to repeat the execution of a particular code block a known number of times.

Often the **range** function is used in conjunction with for loops to create a sequence of numbers from 0 to X, not including X.

```
>>> this_many_times = 5
>>> for i in range(this_many_times):
...     # Do this
...     print('Hurray! ' + str(i))
...
Hurray! 0
Hurray! 1
Hurray! 2
Hurray! 3
Hurray! 4
>>>
```

For Loops

A little more interesting example

It calculates the price of my order from the menu of the pizzeria.

```
# Pizzeria Cashier
menu = {
    'small_pizza': 6,
    'medium_pizza': 12,
    'large_pizza': 18,
    'cookie': 1,
    'soda': 4
}
my_order = ['large_pizza', 'small_pizza']
total_price = 0
for item in my_order:
    # += assignment operator is a shortcut for
    # total_price = total_price + menu[item]
    total_price += menu[item]
print(total_price)
```

While Loops

While loops are generally used when you don’t know how many time you need to repeat the block of code.

Here we’re just showing how the while loop works. It will execute the indented block of code until the condition fails (i.e. until some_number becomes 0)

```
>>> some_number = 7
>>> total = 0
>>> while some_number > 0:
...     total += some_number
...     some_number -= 1
...
>>> print(total)
28
```

Functions

Functions

A function starts with a def keyword followed by the function name. Then in parentheses the arguments of the function.

A DocString is a description of the function is good practice. Can be as long as you like.

In the indented code block you fill in the statements you want to execute.

Optionally you may return something

```
def function_name(argument_1, argument_2):
    """
    Docstring is a description of the function's purpose
    """
    # Do something here
    something = 0
    return something
```

Functions

Function arguments can also have predefined default values, like the **reaction** function here. So you can see that there are default values for **good** and **bad**.

When the reaction function is being called, **good** is being redefined, but the **bad** value isn't being altered.

So if I had a bad test, I would get "Ugh!"

```
>>> def calculate_score(ques):
...     part_1 = (ques[0]/30) * 0.2
...     part_2 = (ques[1]/80) * 0.5
...     part_3 = (ques[2]/40) * 0.3
...     return part_1 + part_2 + part_3
...
>>> def reaction(test, good='Yeah!', bad='Ugh!'):
...     if calculate_score(test) > 0.95:
...         print(good)
...     else:
...         print(bad)
...
>>> my_test = (30, 80, 40)  # in my dreams
>>> reaction(my_test, good=':)')
:)
>>>
```

Objects / Classes

Classes

Classes are objects you define. Using the keyword class followed by the name of the class, then a : colon.

__init__ is the constructor for the class, which is responsible for configuring the newly created class (I won't get into the details).

Output:

<__main__.User object at 0x7f27033cb208>
jonathan.husky@uconn.edu

```
class User:

    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.is_activve = True

    def deactivate(self):
        """ Class method to deactivate user """
        self.is_activve = False
        print("Deactivated " + self.name)

new_user = User('Jonathan', 'jonathan.husky@uconn')
print(new_user)
new_user.email = 'jonathan.husky@uconn.edu'
print(new_user.email)
```

Classes

One of the magic methods is __repr__, which is called when you **print** an instance of the object.

Note, '\n' is a special character that means new line.

Output:

User: Jonathan
active True
jonathan.husky@uconn.edu

```python
class User:

    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.is_activve = True

    def deactivate(self):
        """ Class method to deactivate user """
        self.is_activve = False
        print("Deactivated " + self.name)

    def __repr__(self):
        """ Invoked when printed with print function """
        return("User: " + self.name + '\nactive ' + str(self.is_activve))


new_user = User('Jonathan', 'jonathan.husky@uconn')
print(new_user)
new_user.email = 'jonathan.husky@uconn.edu'
print(new_user.email)
```

Essential Modules

Built-in Essentials

csv: an interface used to parse and create csv files.

math: common mathematical operations, and operations with increase accuracy and precision. “These functions cannot be used with complex numbers; use the functions of the same name from the cmath module if you require support for complex numbers”.

os: “This module provides a portable way of using operating system dependent functionality”.

sys: “This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available”.

json: a library to help parse JavaScript Object Notation data

datetime: provides classes to manipulate dates efficiently and a more user friendly way

Importing Modules

Importing

Using the import keyword you can import modules.

Using the from keyword to specify the module, then the import keyword to specify the specific object from the module.

Best practice is to use just import, unless it's a well known library. This applies particularly for large projects.

```
import os
print(os.path.abspath("./"))

import sys
print(sys.argv)

import math
print(math.pow(5, 2))
print(math.log(64, 2))

from math import sqrt, cos, pi, degrees
print(sqrt(144))
print(cos(pi))
print(degrees(pi))
```

Importing

```
fish  /home/reynaldo/playground/funzone
27 ✓ ls                                          /home/reynaldo/playground/funzone
import_example.py  user_mod.py
28 ✓ cat import_example.py                       /home/reynaldo/playground/funzone
import os
print(os.path.abspath("./"))

import sys
print(sys.argv)

import math
print(math.pow(5, 2))
print(math.log(64, 2))

from math import sqrt, cos, pi, degrees
print(sqrt(144))
print(cos(pi))
print(degrees(pi))
29 ✓ python import_example.py                    /home/reynaldo/playground/funzone
/home/reynaldo/playground/funzone
['import_example.py']
25.0
6.0
12.0
-1.0
180.0
30 ✓                                             /home/reynaldo/playground/funzone
```

Importing

Recall the **User** class we made. I have it here inside the **user_mod.py** file.

Notice what happens when I import it. It runs the code after the class definition. Usually when we import a module that's undesirable behavior.

Preferably we only want this behavior when we run it directly.

```
python  /home/reynaldo/playground/funzone
31 ✓  ls                                              /home/reynaldo/playground/funzone
import_example.py  user_mod.py
32 ✓  cat user_mod.py                                 /home/reynaldo/playground/funzone
class User:

    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.is_activve = True

    def deactivate(self):
        """ Class method to deactivate user """
        self.is_activve = False
        print("Deativated " + self.name)

    def __repr__(self):
        """ Invoked when printed with print function """
        return("User: " + self.name + '\nactive ' + str(self.is_activve))


new_user = User('Johnathan', 'johnathan.husky@uconn')
print(new_user)
new_user.email = 'johnathan.husky@uconn.edu'
print(new_user.email)
33 ✓  python                                          /home/reynaldo/playground/funzone
Python 3.6.3 |Anaconda, Inc.| (default, Nov 20 2017, 20:41:42)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import user_mod
User: Johnathan
active True
johnathan.husky@uconn.edu
>>>
```

Importing

We can curtail this by using the boilerplate:

```
if __name__ == '__main__':
```

which checks if this is the main program or not.

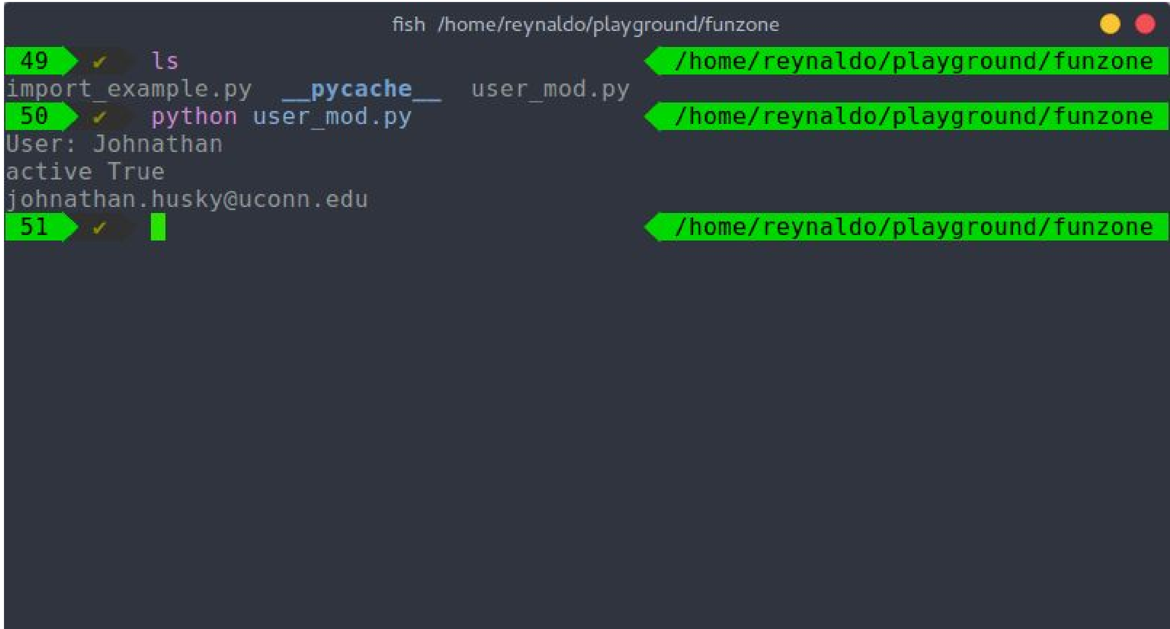
So when we import it will not execute the code inside the boilerplate, since it's not the main program.

```
python  /home/reynaldo/playground/funzone
41  ✔  ls                                          /home/reynaldo/playground/funzone
import_example.py  user_mod.py
42  ✔  cat user_mod.py                             /home/reynaldo/playground/funzone
class User:

    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.is_activve = True

    def deactivate(self):
        """ Class method to deactivate user """
        self.is_activve = False
        print("Deativated " + self.name)

    def __repr__(self):
        """ Invoked when printed with print function """
        return("User: " + self.name + '\nactive ' + str(self.is_activve))

if __name__ == '__main__':
    new_user = User('Johnathan', 'johnathan.husky@uconn')
    print(new_user)
    new_user.email = 'johnathan.husky@uconn.edu'
    print(new_user.email)
43  ✔  python                                      /home/reynaldo/playground/funzone
Python 3.6.3 |Anaconda, Inc.| (default, Nov 20 2017, 20:41:42)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import user_mod
>>> me = user_mod.User('Me', 'me@uconn.edu')
>>> print(me)
User: Me
active True
>>>
```

Importing

You can see that it performs the intended behavior when we execute it directly.

That is, the code inside the boilerplate is executed.



What is __pycache__?

It's a directory holding compiled versions of the programs you execute with python within the directory your are working in.

These compiled files allow for faster execution in future runs, and are recompiled automatically when once run again with changes.

This isn't a thing in python 2.7, but it is in python 3.

Data Science Modules

Essential Data Science Modules

scipy: “It provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimization”.

numpy: “a library for scientific computing including a powerful N-dimensional array object, sophisticated (broadcasting) functions, tools for integrating C/C++ and Fortran code, linear algebra, Fourier transform, and random number capabilities”.

pandas: “library providing high-performance, easy-to-use data structures and data analysis tools”.

matplotlib: “Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.”

plotly: a plotting library for easily creating perhaps the best interactive graphs available today.

jupyter: “The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text”.

Lets install them!

Using environments	
Create a new environment named py35, install Python 3.5	`conda create --name py35 python=3.5`
Activate the new environment to use it	`WINDOWS:    activate py35` `LINUX, macOS: source activate py35`
Get a list of all my environments, active environment is shown with *	`conda env list`
Make exact copy of an environment	`conda create --clone py35 --name py35-2`
List all packages and versions installed in active environment	`conda list`
List the history of each change to the current environment	`conda list --revisions`
Restore environment to a previous revision	`conda install --revision 2`
Save environment to a text file	`conda list --explicit > bio-env.txt`
Delete an environment and everything in it	`conda env remove --name bio-env`
Deactivate the current environment	`WINDOWS: deactivate` `macOS, LINUX: source deactivate`
Create environment from a text file	`conda env create --file bio-env.txt`
Stack commands: create a new environment, name it bio-env and install the biopython package	`conda create --name bio-env biopython`

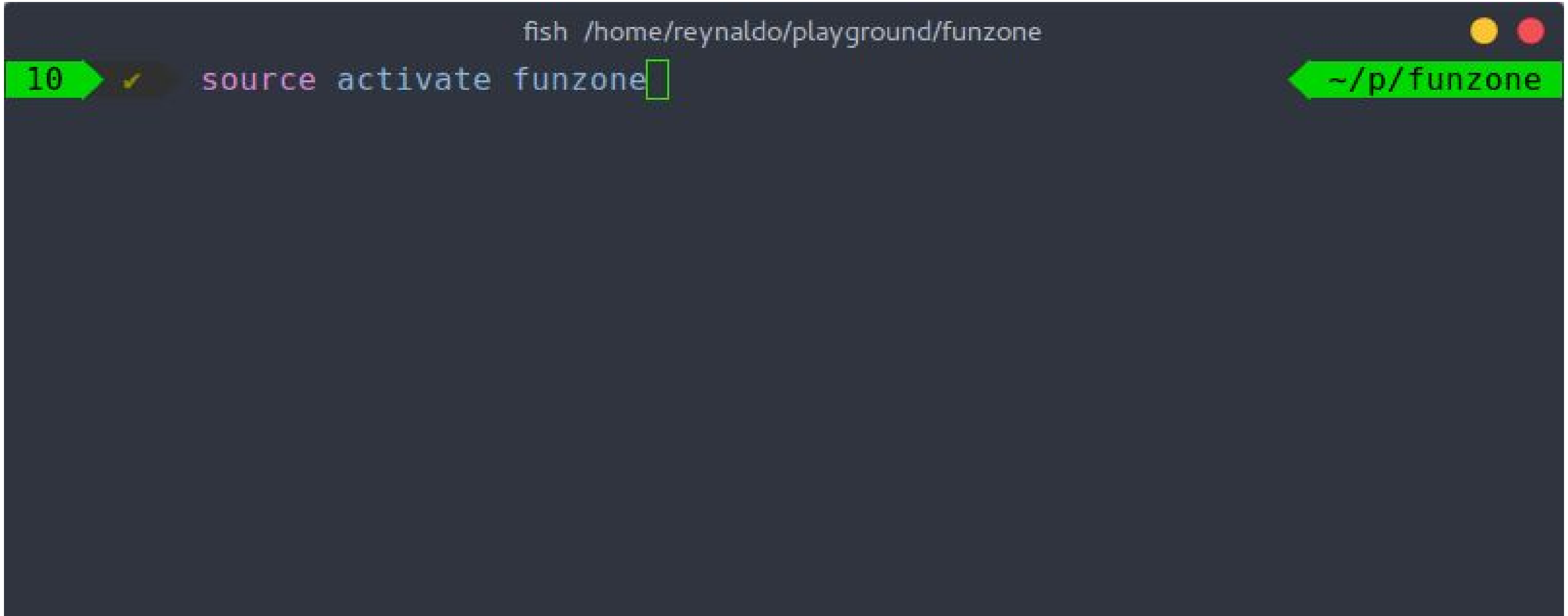
Credits to Conda Cheat Sheet

Working with Conda

Create Environment

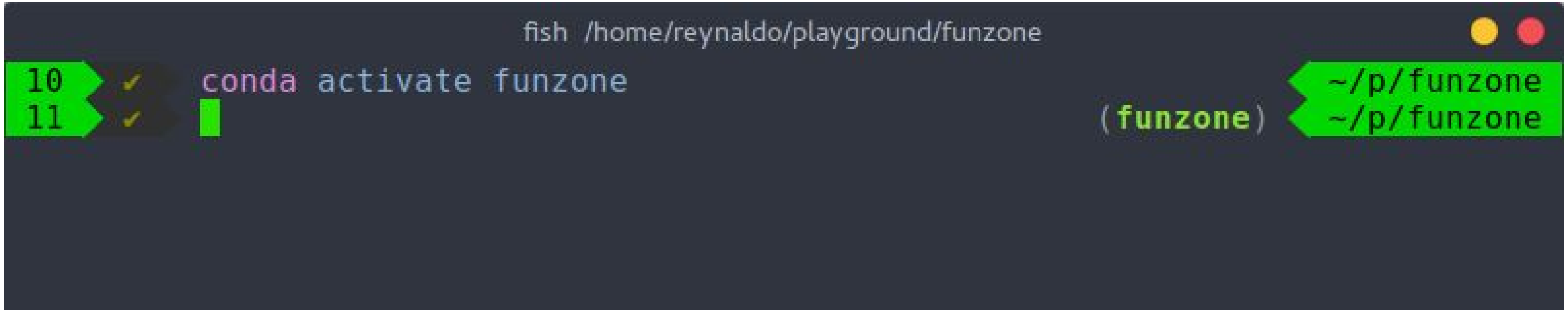
```
fish  /home/reynaldo/playground/funzone
 6  ✔   conda create --name=funzone python=3.6                    ~/p/funzone
Fetching package metadata ...........
Solving package specifications: .

Package plan for installation in environment /home/reynaldo/miniconda3/envs/funz
one:

The following NEW packages will be INSTALLED:

    ca-certificates: 2017.08.26-h1d4fec5_0
    certifi:         2017.11.5-py36hf29ccca_0
    libedit:         3.1-heed3624_0
    libffi:          3.2.1-hd88cf55_4
    libgcc-ng:       7.2.0-h7cc24e2_2
    libstdcxx-ng:    7.2.0-h7a57d05_2
```

Activate Environment



In case you use the fish shell (like me...)

Cannot run source activate with conda in Fish-shell



A few more commands

`conda install PACKAGENAME` Install a package included in Anaconda

`conda search PACKAGENAME` Use conda to search for a package

Anything in conda search can be found on the Anaconda Cloud

`conda update PACKAGENAME` Update a given package

Install Packages

```
fish  /home/reynaldo/playground/funzone
12 ✓  conda install scipy numpy pandas matplotlib plotly jupyter
Fetching package metadata ...........
Solving package specifications: .

Package plan for installation in environment /home/reynaldo/miniconda3/envs/funz
one:

The following NEW packages will be INSTALLED:

    asn1crypto:          0.24.0-py36_0
    bleach:              2.1.2-py36_0
    cffi:                1.11.4-py36h9745a5d_0
    chardet:             3.0.4-py36h0f667ec_1
    cryptography:        2.1.4-py36hd09be54_0
    cycler:              0.10.0-py36h93f1223_0
```

Data Analytics Example Live!

Machine Learning Modules

Popular Machine Learning Modules

scikit-learn: general machine learning, with a user api for creating and testing models quickly, but not optimized for super high performance.

tensorflow: general machine learning, but really built for high performance neural networks.

keras: an interface to tensorflow to make it easier to make deep neural networks

pytorch: like tensorflow, but made by facebook, made purely from python

theano: essentially an extensive high performance math library that is also used to create neural networks.

Machine Learning Example

Here we go again ... breath!