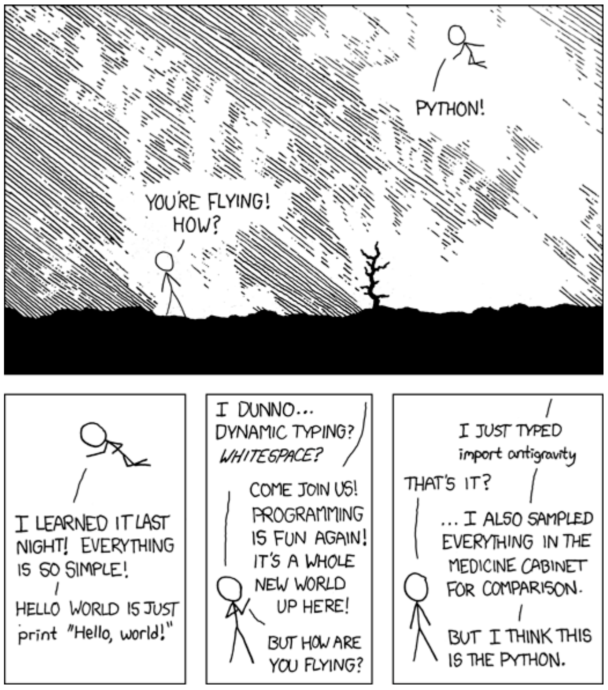
Intro to Python

Why Python?

I. Readibility
II. Built with coder psychology in mind
III. Language interoperability
IV. Object Oriented
V. It does EVERYTHING


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

And simply diving in:

Let's quick look at an example of python showing

- using libraries
- using comments
- variable assignment
- using functions
- printing results

Then we'll go through things more slowly

In [40]:
```
# This is a comment, i'm sure you are familiar with it

# You import libraries (also called packages)
# with the import statement
import math


# You'll notice you don't have to declare the variable type
x = 2 * math.pi
message = 'Math is fun:'
result = math.cos( x )

# Sending the results to the screen is simple with print
print x

# You can print multiple things by separating them with ","
print message, result
```
```
6.28318530718
Math is fun: 1.0
```

Variables

- No need to declare variable types
- ANYTHING can be put in a variable
  - numbers
  - strings
  - functions
  - file-handles
  - etc, etc

Variables are just containers to put stuff in, they don't much care what!

```
In [41]: x = 'good morning'
         print x
```

```
good morning
```

```
In [42]: x = 5
         print x
```

```
5
```

```
In [43]: x = math.cos
         print x( math.pi )
```

```
-1.0
```

Simple data types/containers in python

These are the basic building blocks of any python code. More types are available,
both in standard python and by using additional libraries, which we will go into later.

For now:

- integer
- float
- string
- list

Numbers: ints and floats

```
In [44]: # Integer numbers simply don't have decimal points
         2
```

```
Out[44]: 2
```

```
In [45]: # To make an int a float, just add a decimal
         2.0
```

```
Out[45]: 2.0
```

Excercise:

Do some basic arithmetic with floats and integers.

- addition: +
- subtraction: -
- multiplication: *
- division: /
- exponentiataion: **
- modulo: %

```
In [46]: # Do some basic arithmetic on some numbers
         # mix and match integers and floats and see what happens
```

```
In [47]: print 5+1.0
         print 1.0+1.0
         print 4+5
         print 13/7

6.0
2.0
9
1
```

Strings

```
In [48]: # To say somthing is a string, just enclose it in quotes
         'two'

Out[48]: 'two'
```

```
In [49]: # You can index and slice strings using square brackets
         word = 'hello'
         print word
         print word[1]
         print word[1:4]

hello
e
ell
```

Lists

This is NOT the python equivalent to an IDL array.
This is a list, which serves a different purpose.

```
In [50]: # bulding a list is easy!
         # Just use square brackets: []
         my_list = [1, 5, 7, 4]
```

```
In [51]: # Just like strings, lists can also be indexed and sliced
         print my_list[0]

1
```

```
In [52]: # Lists can have a hetergeneous mix of values
         mixed_list = ['a', 0, 'tree', [1, 2, 3]]
```

Operations...but not only with numbers?

Exercise:

I. Try different operations on different data types: use addition to "add" integers, floats, strings, and lists. Does addition behave the same when acting on a string vs a float?
II. Now try with multiplication, division, and subtraction. Do these operations work on all data types?

If you finish early, take a break

Excercise:

Make a string containing "luminiferous ether" 10 times.

Then make a list of that string 100 times.

In [53]:
```
# This doesn't sound like fun
lum_str = 'luminiferous ether ' * 10
lum_str_big = 'luminiferous ether' * 100

print lum_str
```

```
luminiferous ether luminiferous ether luminiferous ether luminiferous ether luminiferous ether luminiferous ether
luminiferous ether luminiferous ether luminiferous ether luminiferous ether
```

More datatypes

Tuple

In [54]:
```
#Tuple: immutable
my_tuple = (4, 5, 6)
#I can print the array
print my_tuple[1]
```

```
5
```

In [55]:
```
#but can't change it
my_tuple[1] = 10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-55-bf3e082ee24b> in <module>()
      1 #but can't change it
----> 2 my_tuple[1] = 10

TypeError: 'tuple' object does not support item assignment
```

Dictionary

In [ ]:
```
#Dictionaries are sets of keys and values, together they form a tuple called an item
my_dict = {'apple':3, 'banana': 2, 'carrot': 5}

#Dictionaries are unordered:
print my_dict.keys()

#Reference values in a dictionary using keys
print my_dict['apple']
```

In [ ]:
```
#Also, an example of using dictionaries
fruit = 'apple'
#Option 1:
if fruit == 'banana':
    quantity = 2
elif fruit == 'carrot':
    quantity = 5
elif fruit == 'apple':
    quantity = 3
else:
    print 'I do not know how much fruit to by for the fruit you specified'

#Option 2:
    quantity = my_dict[fruit]
```

Numpy Array

In [ ]:
```
#Numpy arrays are often the most intuitive objects.
#These add, multiply, divide, and subtract element by element
import numpy as np
my_array = np.array([1, 3, 6, 5, 3, 8])
my_array_2 = np.arange(6)
print 'my_array', my_array
print 'my_array_2', my_array_2
print my_array + my_array_2
```

Control Statements

All control statements use : followed by indentation to denote that a given statement is acting on everything indented below. This improve readability by forcing you to write good code and remove the chance of forgetting to end your statement

For loop

In [ ]:
```
#Loop over objects
my_list = range(10)
for obj in my_list:
    print obj
```

If Statements

In [ ]:
```
#From above:
if fruit == 'banana':
    quantity = 2
elif fruit == 'carrot':
    quantity = 5
elif fruit == 'apple':
    quantity = 3
else:
    print 'I do not know how much fruit to by for the fruit you specified'
```

Other useful statements:

- while condition:
- break - stop iterating
- continue - go to next interation

Everything is an object

Everthing really is an object in python. Lists, functions, strings, even numbers.

This is important as we start to manipulate these data types.

We can get information about objects and ask them to do things

Attributes: describing an object

In [ ]:
```
my_array = np.ones((4, 6))
print my_array
print my_array.T
print my_array.shape
```

Methods: Doing something with or to an object

In [ ]:
```
#Using the numpy array we created above
print my_array.mean()

#Also max, min ...

#You actually already use an methods when we were learning about dictionaries
print my_dict.keys()
print my_dict.values()
print my_dict.items()
```

What else can you do with an object?

In [ ]:
```
#  You can look at what any object contains with the dir() function
#  This tells the functions (called methods)
#  and simple values/strings (called attributes) that are available.

x = 5
dir( x )
```

In [ ]:
```
# ipython also allows you to view attributes and methods by typing the variable. <tab>
x.
```

Getting Help

In [ ]:
```
#You can get help on anything by using the help function help()
help(math.cos)
```

Excercise:

I. Put your favorite astronomical target in a variable (e.g. astro_obj = 'supernovae')
II. Use the dir() function to find out how to:

- make everything in caps
- count the occurance of each vowel, 'a','e','i','o','u'

In [ ]:
```
astro_obj = 'supernovae'
dir(astro_obj)
astro_obj.count('e')
```

Hand off to JC

Errors and the Traceback

It may look scary, but it is **EXTREMELY** helpful.

When you execute code or run a script, python executes the instructions line by line until it reaches an error or the end of the code.

This means that the interpreter knows exactly where it was when it encountered something that caused a crash, and will tell you exactly which line of code caused the problem.

It also throws useful error messages, which can be useful for de-bugging.

In [ ]:
```
# Let's go back to our Tuple example
my_tuple = (0, 6, 4, 'a')

my_tuple[1] = 5

print my_tuple[3]
```

This doesn't just work for math, it also works for incorrect syntax, missing variables, etc

Exercise

- Find and correct the errors in the cell below until the code runs all the way through without errors

In [ ]:
```
first_name = 'justin
last_name = 'ely'

print first_name, ' ', middle_name, ' ', last_name

age = 25 + 0j
print 'And I am ', age.real_part, ' old'
```

Remember: the traceback is your friend!

Functions

Functions should be:

- *SMALL* When you write code, you should write short functions centered on a single task. Think of your code as an essay and each function as a paragraph. It is overwhelming to read an essay (or book) without paragraphs and you are likely to miss errors you would have otherwise caught. Small functions:
  - Improve readability
  - Easy to reuse
  - avoid repeating code with different variables
  - avoid changing code in multiple places
- *WELL NAMED* Your function calls (and variable names) should describe exactly what you are doing. This makes your code very readable

Let's write a program to read a 2 column text file with number of UFO sightings for each month of a given year (data from: http://www.nuforc.org/webreports/ndxevent.html)

In [ ]:
```
#Start by outlining your code
#I know I'm going to need to read in the information from the file and then parse each line into columns
```

In [ ]:
```
def read_file():
    pass
def parse_file():
    pass
```

In [ ]:
```
def read_file(filename):
    #Open file
    open_file = open(filename, 'r')
    all_lines = open_file.readlines()
    return all_lines
```

In [ ]:
```
def parse_file(all_lines):
    month = []
    number_ufo = []
    for iline in all_lines:
        split_line = iline.split('\t')
        month.append(split_line[0])
        number_ufo.append(int(split_line[1]))
    return month, number_ufo
```

In [ ]:
```
#Now let's build our code, filling in the functions as needed
all_lines = read_file('ufo_sightings_2013.txt')
month, number_ufo = parse_file(all_lines)
```

In [ ]:
```
print month
print number_ufo
```

Exercise

I. Write a function to find the mean of a list of numbers
II. Use read_file, parse_file, and the mean function you just wrote to find the average number of UFO sightings per month in 2013

In [ ]:
```
def calc_mean(my_list):
    total = 0
    for num in my_list:
        total = total + num
    average = float(total) / len(my_list) #Not you have to make one value float or you get rounding due to integer division
    return average
```

In [ ]:
```
all_lines = read_file('ufo_sightings_2013.txt')
month, number_ufo = parse_file(all_lines)
avg_ufo_sightings = calc_mean(number_ufo)
print avg_ufo_sightings
```

Move to text file

- part 1: copy and paste everything in, show that it runs the same from the command line and importing
- part 2: add if **name** == "**main**": show that it runs from the command line but not when imported. Talk about module
- part 3: add in sys.argv to take filename
    - discuss argument order of sys.argv
    - import sys
    - try different files

In [ ]: