13 - Python, Git Branching Wrap-up

CS 2043: Unix Tools and Scripting, Spring 2016 [1]

Stephen McDowell

February 24th, 2016

Cornell University

Table of contents

Some Logistics

- HW2 and `awk`: read Piazza post 144. No floating point arithmetic will be accepted.

Some Logistics

- HW2 and awk: read Piazza post 144. No floating point arithmetic will be accepted.
- Notes on notes, A2.

Some Logistics

- HW2 and `awk`: read Piazza post 144. No floating point arithmetic will be accepted.
- Notes on notes, A2.
- (poll) A2 and its due date, and what our options are.

Some Logistics

- HW2 and awk: read Piazza post 144. No floating point arithmetic will be accepted.
- Notes on notes, A2.
- (poll) A2 and its due date, and what our options are.
    - Can push due date back.

Some Logistics

- HW2 and awk: read Piazza post 144. No floating point arithmetic will be accepted.
- Notes on notes, A2.
- (poll) A2 and its due date, and what our options are.
  - Can push due date back.
  - Have 3 assignments instead of 4, but HW3 will be longer.

Python Overview

What is Python?

- Yet another programming language you can use in your daily hacking.

What is Python?

- Yet another programming language you can use in your daily hacking.
- Extremely convenient, especially where String manipulation and lists are concerned.

What is Python?

- Yet another programming language you can use in your daily hacking.
- Extremely convenient, especially where String manipulation and lists are concerned.
- This all comes at a cost: it is an "interpreted" language.

What is Python?

- Yet another programming language you can use in your daily hacking.
- Extremely convenient, especially where String manipulation and lists are concerned.
- This all comes at a cost: it is an "interpreted" language.
    - In terms of scripting and what you have seen in this class so far, that isn't all that important.

What is Python?

- Yet another programming language you can use in your daily hacking.
- Extremely convenient, especially where String manipulation and lists are concerned.
- This all comes at a cost: it is an "interpreted" language.
    - In terms of scripting and what you have seen in this class so far, that isn't all that important.
    - For the most part, everything we have seen works in a similar way: as a script, it runs top-down.

What is Python?

- Yet another programming language you can use in your daily hacking.
- Extremely convenient, especially where String manipulation and lists are concerned.
- This all comes at a cost: it is an "interpreted" language.
    - In terms of scripting and what you have seen in this class so far, that isn't all that important.
    - For the most part, everything we have seen works in a similar way: as a script, it runs top-down.
    - Raw Python code is "compiled-on-the-fly" as it executes, but you can make it run *very* fast.

What is Python?

- Yet another programming language you can use in your daily hacking.
- Extremely convenient, especially where String manipulation and lists are concerned.
- This all comes at a cost: it is an "interpreted" language.
    - In terms of scripting and what you have seen in this class so far, that isn't all that important.
    - For the most part, everything we have seen works in a similar way: as a script, it runs top-down.
    - Raw Python code is "compiled-on-the-fly" as it executes, but you can make it run *very* fast.
        - Use libraries such as **`numpy`**, **`scipy`**, **`pandas`**, and many more.

What is Python?

- Yet another programming language you can use in your daily hacking.
- Extremely convenient, especially where String manipulation and lists are concerned.
- This all comes at a cost: it is an "interpreted" language.
    - In terms of scripting and what you have seen in this class so far, that isn't all that important.
    - For the most part, everything we have seen works in a similar way: as a script, it runs top-down.
    - Raw Python code is "compiled-on-the-fly" as it executes, but you can make it run *very* fast.
        - Use libraries such as **`numpy`**, **`scipy`**, **`pandas`**, and many more.
        - Write your own **`C`** code and use **`Cython`**.

What is Python?

- Yet another programming language you can use in your daily hacking.
- Extremely convenient, especially where String manipulation and lists are concerned.
- This all comes at a cost: it is an "interpreted" language.
  - In terms of scripting and what you have seen in this class so far, that isn't all that important.
  - For the most part, everything we have seen works in a similar way: as a script, it runs top-down.
  - Raw Python code is "compiled-on-the-fly" as it executes, but you can make it run *very* fast.
    - Use libraries such as `numpy`, `scipy`, `pandas`, and many more.
    - Write your own `C` code and use `Cython`.
    - Doing it on your own is much more challenging, and often has little payoff.

Why Python?

- Easy to test concepts using the interpreter (interactive).

Why Python?

- Easy to test concepts using the interpreter (interactive).
    - I often times will just bust out the interpreter if I need to check some math really quickly.

Why Python?

- Easy to test concepts using the interpreter (interactive).
  - I often times will just bust out the interpreter if I need to check some math really quickly.
- You have the ability to write object-oriented code if you want, but you can also just write scripts.

Why Python?

- Easy to test concepts using the interpreter (interactive).
  - I often times will just bust out the interpreter if I need to check some math really quickly.
- You have the ability to write object-oriented code if you want, but you can also just write scripts.
- Wide range of built-in functions to accomplish pretty much everything you want to do:

Why Python?

- Easy to test concepts using the interpreter (interactive).
    - I often times will just bust out the interpreter if I need to check some math really quickly.
- You have the ability to write object-oriented code if you want, but you can also just write scripts.
- Wide range of built-in functions to accomplish pretty much everything you want to do:
    - Create ranges of numbers easily.

Why Python?

- Easy to test concepts using the interpreter (interactive).
    - I often times will just bust out the interpreter if I need to check some math really quickly.
- You have the ability to write object-oriented code if you want, but you can also just write scripts.
- Wide range of built-in functions to accomplish pretty much everything you want to do:
    - Create ranges of numbers easily.
    - Generate random numbers with ease.

Why Python?

- Easy to test concepts using the interpreter (interactive).
    - I often times will just bust out the interpreter if I need to check some math really quickly.
- You have the ability to write object-oriented code if you want, but you can also just write scripts.
- Wide range of built-in functions to accomplish pretty much everything you want to do:
    - Create ranges of numbers easily.
    - Generate random numbers with ease.
    - Write to and read from files like no other language can.

Why Python?

- Easy to test concepts using the interpreter (interactive).
    - I often times will just bust out the interpreter if I need to check some math really quickly.
- You have the ability to write object-oriented code if you want, but you can also just write scripts.
- Wide range of built-in functions to accomplish pretty much everything you want to do:
    - Create ranges of numbers easily.
    - Generate random numbers with ease.
    - Write to and read from files like no other language can.
    - Honestly, there are way too many examples. Python is great!

Why Python?

- Easy to test concepts using the interpreter (interactive).
  - I often times will just bust out the interpreter if I need to check some math really quickly.
- You have the ability to write object-oriented code if you want, but you can also just write scripts.
- Wide range of built-in functions to accomplish pretty much everything you want to do:
  - Create ranges of numbers easily.
  - Generate random numbers with ease.
  - Write to and read from files like no other language can.
  - Honestly, there are way too many examples. Python is great!
- Easy to play with: just type **`python`** and hit enter to bring up the interpreter.

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.
- `str`: strings. Can use `'single quotes'` or `"double quotes"`.

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.
- `str`: strings. Can use `'single quotes'` or `"double quotes"`.
- `list`: uses brackets:

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.
- `str`: strings. Can use `'single quotes'` or `"double quotes"`.
- `list`: uses brackets:

```
    [0,3,5]
```

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.
- `str`: strings. Can use `'single quotes'` or `"double quotes"`.
- `list`: uses brackets:

```
    [0,3,5]
    ["mixed",3.4,"data",False,"types"]
```

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.
- `str`: strings. Can use `'single quotes'` or `"double quotes"`.
- `list`: uses brackets:

```
    [0,3,5]
    ["mixed",3.4,"data",False,"types"]
```

- `tuple`: use parenthesis:

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.
- `str`: strings. Can use `'single quotes'` or `"double quotes"`.
- `list`: uses brackets:

```
    [0,3,5]
    ["mixed",3.4,"data",False,"types"]
```

- `tuple`: use parenthesis:

```
    (0,3,5)
```

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.
- `str`: strings. Can use `'single quotes'` or `"double quotes"`.
- `list`: uses brackets:

```
[0,3,5]
["mixed",3.4,"data",False,"types"]
```

- `tuple`: use parenthesis:

```
(0,3,5)
("mixed",3.4,"data",False,"types")
```

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.
- `str`: strings. Can use `'single quotes'` or `"double quotes"`.
- `list`: uses brackets:
  ```
  [0,3,5]
  ["mixed",3.4,"data",False,"types"]
  ```
- `tuple`: use parenthesis:
  ```
  (0,3,5)
  ("mixed",3.4,"data",False,"types")
  ```
- `dict`: map keys and values

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.
- `str`: strings. Can use `'single quotes'` or `"double quotes"`.
- `list`: uses brackets:
```
[0,3,5]
["mixed",3.4,"data",False,"types"]
```
- `tuple`: use parenthesis:
```
(0,3,5)
("mixed",3.4,"data",False,"types")
```
- `dict`: map keys and values
```
{"a" : 1, 3 : "three", 3.999 : "four"}
```

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.
- `str`: strings. Can use `'single quotes'` or `"double quotes"`.
- `list`: uses brackets:

```
[0,3,5]
["mixed",3.4,"data",False,"types"]
```

- `tuple`: use parenthesis:

```
(0,3,5)
("mixed",3.4,"data",False,"types")
```

- `dict`: map keys and values

```
{"a" : 1, 3 : "three", 3.999 : "four"}
```

- CAUTION: strings and tuples cannot be changed. EVER.

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.
- `str`: strings. Can use `'single quotes'` or `"double quotes"`.
- `list`: uses brackets:

  ```
  [0,3,5]
  ["mixed",3.4,"data",False,"types"]
  ```
- `tuple`: use parenthesis:

  ```
  (0,3,5)
  ("mixed",3.4,"data",False,"types")
  ```
- `dict`: map keys and values

  ```
  {"a" : 1, 3 : "three", 3.999 : "four"}
  ```
- CAUTION: strings and tuples cannot be changed. EVER.
  - Constantly "updating" the value of a string?

The Quick and Dirty Basics

- `bool`: boolean vales (`True` and `False` - capital first letter).
- `int`: whole numbers.
- `float`: decimal numbers.
- `str`: strings. Can use `'single quotes'` or `"double quotes"`.
- `list`: uses brackets:
```
[0,3,5]
["mixed",3.4,"data",False,"types"]
```
- `tuple`: use parenthesis:
```
(0,3,5)
("mixed",3.4,"data",False,"types")
```
- `dict`: map keys and values
```
{"a" : 1, 3 : "three", 3.999 : "four"}
```
- CAUTION: strings and tuples cannot be changed. EVER.
  - Constantly "updating" the value of a string?
  - You are making new strings every single time.

Indexing Items

- Starts at `0`, goes up to but *not including* `len(item)`.

Indexing Items

- Starts at `0`, goes up to but *not including* `len(item)`.
- Can use splicing to grab ranges of valid indices!

Indexing Items

- Starts at `0`, goes up to but *not including* `len(item)`.
- Can use splicing to grab ranges of valid indices!

```
>>> a_list = [1, 2, 3, 4, 5, 6]
>>> a_list[0]
1
>>> a_list[-1]
6
>>> len(a_list)
6
>>> a_list[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> a_list[1:4]
[2, 3, 4]
```

Indexing Items

- Starts at `0`, goes up to but *not including* `len(item)`.
- Can use splicing to grab ranges of valid indices!

```
>>> a_list = [1, 2, 3, 4, 5, 6]
>>> a_list[0]
1
>>> a_list[-1]
6
>>> len(a_list)
6
>>> a_list[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> a_list[1:4]
[2, 3, 4]
```

```
>>> a_string = "123456"
>>> a_string[0]
'1'
>>> a_string[-1]
'6'
>>> len(a_string)
6
>>> a_string[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> a_string[1:4]
'234'
```

Working with Lists

- Reverse a list with `list_variable.reverse()`

Working with Lists

- Reverse a list with `list_variable.reverse()`
- Append to a list with `list_variable.append(item)`

Working with Lists

- Reverse a list with `list_variable.reverse()`
- Append to a list with `list_variable.append(item)`
- Sort a list with `list_variable.sort()`

Working with Lists

- Reverse a list with `list_variable.reverse()`
- Append to a list with `list_variable.append(item)`
- Sort a list with `list_variable.sort()`
- Get a separate sorted list without changing the original list: `new_list = sorted(orig_list)`

Working with Lists

- Reverse a list with `list_variable.reverse()`
- Append to a list with `list_variable.append(item)`
- Sort a list with `list_variable.sort()`
- Get a separate sorted list without changing the original list: `new_list = sorted(orig_list)`
- Find the index of an item with `list_variable.index(item)`

Working with Lists

- Reverse a list with `list_variable.reverse()`
- Append to a list with `list_variable.append(item)`
- Sort a list with `list_variable.sort()`
- Get a separate sorted list without changing the original list: `new_list = sorted(orig_list)`
- Find the index of an item with `list_variable.index(item)`
- Retrieve the last element with `list_variable.pop()`

Working with Lists

- Reverse a list with `list_variable.reverse()`
- Append to a list with `list_variable.append(item)`
- Sort a list with `list_variable.sort()`
- Get a separate sorted list without changing the original list: `new_list = sorted(orig_list)`
- Find the index of an item with `list_variable.index(item)`
- Retrieve the last element with `list_variable.pop()`
    - You can emulate stacks and queues easily with python lists.

Working with Lists

- Reverse a list with `list_variable.reverse()`
- Append to a list with `list_variable.append(item)`
- Sort a list with `list_variable.sort()`
- Get a separate sorted list without changing the original list: `new_list = sorted(orig_list)`
- Find the index of an item with `list_variable.index(item)`
- Retrieve the last element with `list_variable.pop()`
    - You can emulate stacks and queues easily with python lists.
- Get the documentation with `help(list)`

Working with Dictionaries

- Declare in-line:

Working with Dictionaries

- Declare in-line:
    - `d = {'key1' : 'val1', 'key2' : 'val2'}`

Working with Dictionaries

- Declare in-line:
  - `d = {'key1' : 'val1', 'key2' : 'val2'}`
- Declare an empty dictionary: **`d = {}`**

Working with Dictionaries

- Declare in-line:
    - `d = {'key1' : 'val1', 'key2' : 'val2'}`
- Declare an empty dictionary: `d = {}`
- Get the list of keys with `d.keys()`

Working with Dictionaries

- Declare in-line:
  - `d = {'key1' : 'val1', 'key2' : 'val2'}`
- Declare an empty dictionary: `d = {}`
- Get the list of keys with `d.keys()`
- Get the list of values with `d.values()`

Working with Dictionaries

- Declare in-line:
    - `d = {'key1' : 'val1', 'key2' : 'val2'}`
- Declare an empty dictionary: `d = {}`
- Get the list of keys with `d.keys()`
- Get the list of values with `d.values()`
- Can loop through dictionaries with ease:

Working with Dictionaries

- Declare in-line:
  - `d = {'key1' : 'val1', 'key2' : 'val2'}`
- Declare an empty dictionary: `d = {}`
- Get the list of keys with `d.keys()`
- Get the list of values with `d.values()`
- Can loop through dictionaries with ease:

```
>>> d = {'key1' : 'val1', 'key2' : 'val2'}
>>> for k, v in d.iteritems(): # replaced by items() in py3
        print(k, v)
    key2 val2
    key1 val1
```

Working with Dictionaries

- Declare in-line:
  - `d = {'key1' : 'val1', 'key2' : 'val2'}`
- Declare an empty dictionary: `d = {}`
- Get the list of keys with `d.keys()`
- Get the list of values with `d.values()`
- Can loop through dictionaries with ease:

```
>>> d = {'key1' : 'val1', 'key2' : 'val2'}
>>> for k, v in d.iteritems(): # replaced by items() in py3
        print(k, v)
    key2 val2
    key1 val1
```

- Add / overwrite items with `d['key'] = 'value'`

Python is Powerful *and* Flexible

- By now you have heard me say something along the lines of " if it's more than 10 lines I just do it in Python".

Python is Powerful *and* Flexible

- By now you have heard me say something along the lines of " if it's more than 10 lines I just do it in Python".
- Here is why:

Python is Powerful *and* Flexible

- By now you have heard me say something along the lines of " if it's more than 10 lines I just do it in Python".
- Here is why:
    - You have access to your favorite `POSIX` sets.

Python is Powerful *and* Flexible

- By now you have heard me say something along the lines of "if it's more than 10 lines I just do it in Python".
- Here is why:
    - You have access to your favorite `POSIX` sets.
    - Formatting strings is superbly powerful.

Python is Powerful *and* Flexible

- By now you have heard me say something along the lines of "if it's more than 10 lines I just do it in Python".
- Here is why:
    - You have access to your favorite `POSIX` sets.
    - Formatting strings is superbly powerful.
    - Did I mention Python has probably the best documentation of any language *ever*?

Python is Powerful *and* Flexible

- By now you have heard me say something along the lines of "if it's more than 10 lines I just do it in Python".
- Here is why:
  - You have access to your favorite `POSIX` sets.
  - Formatting strings is superbly powerful.
  - Did I mention Python has probably the best documentation of any language *ever*?
  - Very easy to get something out quick and correct.

Python is Powerful *and* Flexible

- By now you have heard me say something along the lines of "if it's more than 10 lines I just do it in Python".
- Here is why:
    - You have access to your favorite `POSIX` sets.
    - Formatting strings is superbly powerful.
    - Did I mention Python has probably the best documentation of any language *ever*?
    - Very easy to get something out quick and correct.
    - Loops are flexible and easy.

Python is Powerful *and* Flexible

- By now you have heard me say something along the lines of "if it's more than 10 lines I just do it in Python".
- Here is why:
    - You have access to your favorite `POSIX` sets.
    - Formatting strings is superbly powerful.
    - Did I mention Python has probably the best documentation of any language *ever*?
    - Very easy to get something out quick and correct.
    - Loops are flexible and easy.
    - Functional if you need it (`lambda`).

Python is Powerful *and* Flexible

- By now you have heard me say something along the lines of "if it's more than 10 lines I just do it in Python".
- Here is why:
  - You have access to your favorite `POSIX` sets.
  - Formatting strings is superbly powerful.
  - Did I mention Python has probably the best documentation of any language *ever*?
  - Very easy to get something out quick and correct.
  - Loops are flexible and easy.
  - Functional if you need it (`lambda`).
    - Extraordinary type-checking system. Use `type(variable)` to see what it *currently* is.

Python is Powerful *and* Flexible

- By now you have heard me say something along the lines of "if it's more than 10 lines I just do it in Python".
- Here is why:
    - You have access to your favorite `POSIX` sets.
    - Formatting strings is superbly powerful.
    - Did I mention Python has probably the best documentation of any language *ever*?
    - Very easy to get something out quick and correct.
    - Loops are flexible and easy.
    - Functional if you need it (`lambda`).
        - Extraordinary type-checking system. Use `type(variable)` to see what it *currently* is.
    - Exception handling is easy.

Python is Powerful *and* Flexible

- By now you have heard me say something along the lines of "if it's more than 10 lines I just do it in Python".
- Here is why:
  - You have access to your favorite `POSIX` sets.
  - Formatting strings is superbly powerful.
  - Did I mention Python has probably the best documentation of any language *ever*?
  - Very easy to get something out quick and correct.
  - Loops are flexible and easy.
  - Functional if you need it (`lambda`).
    - Extraordinary type-checking system. Use `type(variable)` to see what it *currently* is.
  - Exception handling is easy.
  - Executing system calls (e.g. unix shell commands) is as easy as importing a module.

Python is Powerful *and* Flexible

- By now you have heard me say something along the lines of "if it's more than 10 lines I just do it in Python".
- Here is why:
    - You have access to your favorite `POSIX` sets.
    - Formatting strings is superbly powerful.
    - Did I mention Python has probably the best documentation of any language *ever*?
    - Very easy to get something out quick and correct.
    - Loops are flexible and easy.
    - Functional if you need it (`lambda`).
        - Extraordinary type-checking system. Use `type(variable)` to see what it *currently* is.
    - Exception handling is easy.
    - Executing system calls (e.g. unix shell commands) is as easy as importing a module.
    - Operator overloading: read Kenneth Love's article in [2]. Many other excellent resources from him linked on that site.

Scripting with Python

Writing Python Code

- PYTHON 3 IS NOT COMPATIBLE WITH PYTHON 2.

Writing Python Code

- PYTHON 3 IS NOT COMPATIBLE WITH PYTHON 2.
    - Easy example:

Writing Python Code

- PYTHON 3 IS NOT COMPATIBLE WITH PYTHON 2.
  - Easy example:
  - Python 2 (no need for parentheses): `print "a", "b", "c"`

Writing Python Code

- PYTHON 3 IS NOT COMPATIBLE WITH PYTHON 2.
    - Easy example:
    - Python 2 (no need for parentheses): `print "a", "b", "c"`
    - Python 3 (need parentheses): `print("a", "b", "c")`

Writing Python Code

- PYTHON 3 IS NOT COMPATIBLE WITH PYTHON 2.
    - Easy example:
    - Python 2 (no need for parentheses): `print "a", "b", "c"`
    - Python 3 (need parentheses): `print("a", "b", "c")`
- Python is parsed by white-space:

Writing Python Code

- PYTHON 3 IS NOT COMPATIBLE WITH PYTHON 2.
    - Easy example:
    - Python 2 (no need for parentheses): `print "a", "b", "c"`
    - Python 3 (need parentheses): `print("a", "b", "c")`
- Python is parsed by white-space:
    - Bad indentation will either lead to unexpected results, or program crash.

Writing Python Code

- PYTHON 3 IS NOT COMPATIBLE WITH PYTHON 2.
  - Easy example:
  - Python 2 (no need for parentheses): `print "a", "b", "c"`
  - Python 3 (need parentheses): `print("a", "b", "c")`
- Python is parsed by white-space:
  - Bad indentation will either lead to unexpected results, or program crash.
- In terms of the shebang:

Writing Python Code

- PYTHON 3 IS NOT COMPATIBLE WITH PYTHON 2.
    - Easy example:
    - Python 2 (no need for parentheses): `print "a", "b", "c"`
    - Python 3 (need parentheses): `print("a", "b", "c")`
- Python is parsed by white-space:
    - Bad indentation will either lead to unexpected results, or program crash.
- In terms of the shebang:
    - Writing python2 code: `#!/usr/bin/env python`

Writing Python Code

- PYTHON 3 IS NOT COMPATIBLE WITH PYTHON 2.
    - Easy example:
    - Python 2 (no need for parentheses): `print "a", "b", "c"`
    - Python 3 (need parentheses): `print("a", "b", "c")`
- Python is parsed by white-space:
    - Bad indentation will either lead to unexpected results, or program crash.
- In terms of the shebang:
    - Writing python2 code: `#!/usr/bin/env python`
    - Writing python3 code: `#!/usr/bin/env python3`

Be Aware

- The `xrange` function in Python 2 will prevent you from crashing on large lists.

Be Aware

- The `xrange` function in Python 2 will prevent you from crashing on large lists.
    - Regular **`range`** will create an entire list first.

Be Aware

- The `xrange` function in Python 2 will prevent you from crashing on large lists.
    - Regular `range` will create an entire list first.
- The `xrange` functionality of iteration instead of list entirely replaced `range` in python 3, so `xrange` does not exist anymore (just `range`)!

Be Aware

- The `xrange` function in Python 2 will prevent you from crashing on large lists.
  - Regular `range` will create an entire list first.
- The `xrange` functionality of iteration instead of list entirely replaced `range` in python 3, so `xrange` does not exist anymore (just `range`)!
- There are many other important differences between python 2 and python 3.

Be Aware

- The `xrange` function in Python 2 will prevent you from crashing on large lists.
  - Regular `range` will create an entire list first.
- The `xrange` functionality of iteration instead of list entirely replaced `range` in python 3, so `xrange` does not exist anymore (just `range`)!
- There are many other important differences between python 2 and python 3.
- In terms of comparisons:

Be Aware

- The `xrange` function in Python 2 will prevent you from crashing on large lists.
    - Regular `range` will create an entire list first.
- The `xrange` functionality of iteration instead of list entirely replaced `range` in python 3, so `xrange` does not exist anymore (just `range`)!
- There are many other important differences between python 2 and python 3.
- In terms of comparisons:
    - The `==` operation calls `__eq__` (if defined), which is *value* comparison.

Be Aware

- The `xrange` function in Python 2 will prevent you from crashing on large lists.
    - Regular `range` will create an entire list first.
- The `xrange` functionality of iteration instead of list entirely replaced `range` in python 3, so `xrange` does not exist anymore (just `range`)!
- There are many other important differences between python 2 and python 3.
- In terms of comparisons:
    - The `==` operation calls `__eq__` (if defined), which is *value* comparison.
    - Comparing `id(var1) == id(var2)` with the `is` keyword does *reference* comparison (are these two literally the same thing in memory).

Be Aware

- The `xrange` function in Python 2 will prevent you from crashing on large lists.
  - Regular `range` will create an entire list first.
- The `xrange` functionality of iteration instead of list entirely replaced `range` in python 3, so `xrange` does not exist anymore (just `range`)!
- There are many other important differences between python 2 and python 3.
- In terms of comparisons:
  - The `==` operation calls `__eq__` (if defined), which is *value* comparison.
  - Comparing `id(var1) == id(var2)` with the `is` keyword does *reference* comparison (are these two literally the same thing in memory).
  - Extremely important to know the difference for `str` objects, since strings are immutable. Example on next slide.

String Comparison

```
#!/usr/bin/env python
#              ^^^^^^ could be python3 too...
# Define a simple function to print various relations...
# ...string formating is really convenient!
def eval(s1, s2):
    print("'{0}' == '{1}': {2}".format(s1,s2,s1 == s2))
    print("'{0}' is '{1}': {2}".format(s1,s2,s1 is s2))
    print("  id(s1): {0}".format(id(s1)))
    print("  id(s2): {0}".format(id(s2)))
#
# Make some strings and evaluate them...
#
x = "dog"
y = "cat"
eval(x,y)
#
# Change the strings and evaluate again...
#
print("\n...change value of y to dog...\n")
# This may seem like a crazy way to make "dog", but
# Python is smart enough to know that if you say
#
#     y = "dog"
#
# Given that x is already that value, it will point
# to it instead of making a new one.  This just forces
# the creation of a new string.
y = ''.join(char for char in ['d','o','g'])
eval(x,y)
```

Working with Files

- Although there are other ways to open files, you should do this way:

Working with Files

- Although there are other ways to open files, you should do this way:

```
with open("filename", "r") as f:
    for line in f:
        print(line)
```

Working with Files

- Although there are other ways to open files, you should do this way:

```
with open("filename", "r") as f:
    for line in f:
        print(line)
```

- There are different file modes, e.g. `r` is read, `w` is write (but will overwrite if the file exists).

Working with Files

- Although there are other ways to open files, you should do this way:

```
with open("filename", "r") as f:
    for line in f:
        print(line)
```

- There are different file modes, e.g. `r` is read, `w` is write (but will overwrite if the file exists).
- The **`with`** statement in python is pure magic, and if your code crashes for whatever reason python will go through and close the file properly for you. There are many other cases you will find the **`with`** statement superior, e.g. with **`threads`** and **`locks`** (CS 4410).

Git Branching Wrap-up

Lets Make our Own Branches

Let's make our own feature branch:

https://github.com/cs2043-sp16/lecture-demos/tree/master/lec13

References I

[1] B. Abrahao, H. Abu-Libdeh, N. Savva, D. Slater, and others over the years.
Previous cornell cs 2043 course slides.

[2] K. Love.
Operator overloading in python.
`http://blog.teamtreehouse.com/operator-overloading-python`.