Python Tutorial

A Gentle Introduction 2

Yann Tambouret

Scientific Computing and Visualization
Information Services & Technology
Boston University
111 Cummington St.
yannpaul@bu.edu

October 2012

BOSTON UNIVERSITY

This Tutorial

- This tutorial is for someone with any programming experience.
- First a brief recap of
  variables, types, if-statement, and functions
- Then I'll cover the basics of
  1. lists
  2. loops
  3. dictionaries
  4. modules

BOSTON UNIVERSITY

Python Overview

- Python is named after the BBC show "Monty Python's Flying Circus"
- We will focus on Python 2 today.
- Python on Katana and on Windows or Mac
- This tutorial borrows largely from a tutorial by the Boston Python Group

BOSTON UNIVERSITY

Python is Interpreted

- Python can be run interactively.
- Code $\Rightarrow$ execution is almost instant; No explicit compilation step required.
- This allows for a faster development process
- The final product is usually more resource intensive, and as a side effect slower then comparable C/Fortran code.

BOSTON UNIVERSITY

Python is Interactive

Practice running python, type **python** in the terminal, hit Enter:

```
1 % python
2 Python 2.7 (#1, Feb 28 2010, 00:02:06)
3 Type "help", "copyright", "credits" or "license" for more i
4 >>>
```

- The '>>>' is a prompt asking for the next python line of code.
- Sometimes you'll see '...' as a prompt too.
- To exit, type `exit()` and Enter
  Try it!

Interactive Python

- Use up/down arrows to navigate history of commands.
- Python can be used as a calculator:

```
1 >>> 7*9
2 63
3 >>> 1/2 + 10
4 10
5 >>> 1.0 / 2 + 20 / 2.0
6 10.5
```

`type()` and Variables

- `type()` returns what Python thinks something is.
- You can store information in *variables*
- Variable names can't have spaces, must start with a letter.
- Don't need to declare a type

```
1 >>> type(1)
2 <type 'int'>
3 >>> x = 2.0
4 >>> type(x)
5 <type 'float'>
```

About Numbers

- A Python **Int** corresponds to **C** `long`, usually...
- A python **Float** is traditionally a **C** `double`
- **Complex** is type that corresponds to a real and imaginary float
- And there are other standard types

Strings

- `'Hello'` or `"Hello"`
- But don't forget to escape:
  Yes: `'I\'m a happy camper'`
  Yes: `"I'm a happy camper"`
  No: `'I'm a happy camper'`
- Concatenation: `"Hello "+ "World"`
- `print` sends string to standard output
- `"Hello"+ 1` results in an error: `TypeError`
- `"Yes" * 3` allowed, repeats: `'YesYesYes'`
  `"3"*3` does not become `"9"` or `9`

BOSTON UNIVERSITY

Booleans

- A type the corresponds to `True` or `False`
  `type(True)` results in `<type 'bool'>`.
- Can compare values to create a Boolean: `1 == 1` is `True`
- Also available: `!=`, $<$, $>$, $<=$, $>=$
- Watch out for `=` and `==` mistakes
- `in` allows you to test if something is in a list:
  `"H" in "Hello"` is `True`
- `not` let's you do the opposite:
  `"Perl" not in "Python Tutorial"` is also `True`

if, elif and else

```
1 sister_age = 15
2 brother_age = 12
3 if sister_age > brother_age:
4     print "sister is older"
5 elif sister_age == brother_age:
6     print "sister and brother are the same age"
7 else:
8     print "brother is older"
```

- `if` takes a Boolean, and executes code if `True`
- `elif` is an alternative tested if the first case is `False`
- `else` occurs if all previous fail.
- And indentation is important!

Indentation, what's up with that?

- If you've programmed in other languages, this indentation thing might seem weird.
- Python prides itself as an easy-to-read language, and indentation makes it easy to read **code blocks**.
- So Python requires indentation over if/end-if, begin-block/end-block organization.

BOSTON UNIVERSITY

Indentation - example

```
1 # this looks like other languages,
2 # but I use a comment to organize
3 if 1 == 1:
4     print "Everything is going to be OK!"
5     if 10 < 0:
6         print "or is it?"
7     #end if
8     print "Inside first code block!"
9 #end if
```

Don't use `#end if`, just keep it in your mind if it gets confusing...

Functions

Defining code for re-use:

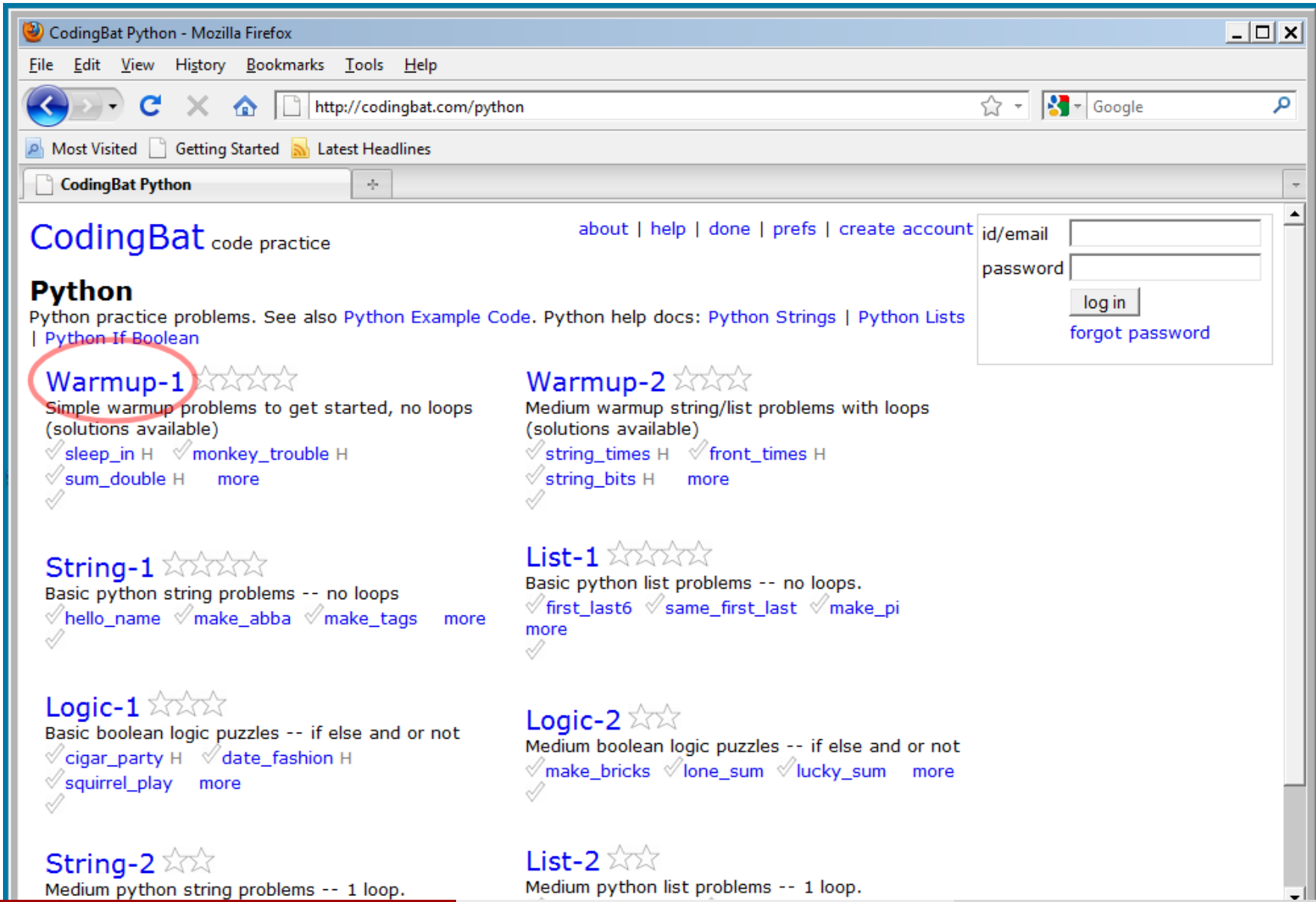
1. Declare signature using `def`:

   ```
   def myFunction(name, age, hair_color):
   ```

2. Add a useful **code block**; indent!
3. Return a result using the `return` statement

```
1 def add(x, y):
2     return x + y
3
4 result = add(1234, 5678)
5 print result
6 result = add(-1.5, .5)
7 print result
```

http://codingbat.com/python



Python as a Scripting Language

You can store statements in a file, and pass that file to Python for execution.
See for example `examples/nobel.py`.

1. How do you comment code in Python?
2. How do you print a newline?
3. How do you print a multi-line string so that whitespace is preserved?

BOSTON UNIVERSITY

Lists - Initialization

In it's simplest form, a list is a comma-separated list of values surrounded by square brackets [and]:

```
1 alist = ['a', 'b', 'c']
2 alist
3 [1, 2, 3]
4 [1, 2.0, '3']
```

Try this out ... results in ...

```
1 >>> alist = ['a', 'b', 'c']
2 >>> alist
3 ['a', 'b', 'c']
4 >>> [1, 2, 3]
5 [1, 2, 3]
6 >>> [1, 2.0, '3']
7 [1, 2.0, '3']
```

BOSTON UNIVERSITY

len() and Access

`len()` is a list method that returns the length of the list:

```
1 >>> len(alist)
2 3
```

We can access elements starting at 0, ending at `len(alist)-1`:

```
1 print alist[0], alist[1], alist[2]
```

results in ...

```
1 >>> print alist[0], alist[1], alist[2]
2 a b c
```

Access

And negative indicies access the elements in reverse order, from -1 to `-len(list)`:

```
1 >>> print alist[-1], alist[-2], alist[-3]
2 c b a
```

And we can use list values as if they were single value variables:

```
1 >>> print "I am " + alist[0] + " list"
2 I am a list
```

Dot-product

Look at `practice\dot_product1.py`
and try the assignment and I'll show the solution next.

BOSTON UNIVERSITY

Modifying a List

You can `append()` a list:

```
1 >>> alist.append('d')
2 >>> alist
3 ['a', 'b', 'c', 'd']
```

You can concatenate lists by "adding" them:

```
1 >>> new_alist = alist + ['e', 'g']
2 >>> new_alist
3 ['a', 'b', 'c', 'd', 'e', 'g']
```

You can also `insert(pos, value)`:

```
1 >>> new_alist.insert(5, 'f')
2 >>> print new_alist
3 ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

BOSTON UNIVERSITY

Modifying a List

You can also change individual elements of a list:

```
1 >>> new_alist[2] = 'C'
2 >>> print new_alist
3 ['a', 'b', 'C', 'd', 'e', 'f', 'g']
```

Slice

You can *slice* a list, access a portion of it.

```
1 >>> new_alist[1:3]
2 ['b', 'C']
```

This gets a new list from the 1 index (inclusive) to the 3 index (exclusive).
So [1,3)
Now try (explore) some other options:

```
1 new_alist = ['a', 'b', 'C', 'd', 'e', 'f', 'g']
2 new_alist[:]
3 new_alist[-3:-1]
4 new_alist[1:]
5 new_alist[:4]
```

BOSTON UNIVERSITY

Slice - Output

```
1 >>> new_alist = ['a', 'b', 'C', 'd', 'e', 'f', 'g']
2 >>> new_alist[:]
3 ['a', 'b', 'C', 'd', 'e', 'f', 'g']
4 >>> new_alist[-3:-1]
5 ['e', 'f']
6 >>> new_alist[1:]
7 ['b', 'C', 'd', 'e', 'f', 'g']
8 >>> new_alist[:4]
9 ['a', 'b', 'C', 'd']
```

BOSTON UNIVERSITY

Modifying by Slicing

You can update many values at once with *slicing*.
What happens when you try:

```
1 >>> new_alist[0:2] = ['A', 'B']
2 >>> print new_alist
3 ['A', 'B', 'C', 'd', 'e', 'f', 'g']
```

Deleting

You can delete an element too, using `del`. Try:

```
1 del new_alist[5]
2 print new_alist
```

```
1 >>> del new_alist[5]
2 >>> print new_alist
3 ['A', 'B', 'C', 'd', 'e', 'g']
```

Strings and Lists

Strings can act like lists:

```
1 >>> name = "Tommy"
2 >>> name[0:3]
3 'Tom'
4 >>> len(name)
5 5
```

But Strings can't be changed like lists:

```
1 >>> name[1] = 'a'
2 Traceback (most recent call last):
3   File "<console>", line 1, in <module>
4 TypeError: 'str' object does not support item assignment
```

BOSTON UNIVERSITY

Loops - For Loop

`for` is used to repeat a task on each element in a list:

```
1 a = ['cat', 'window', 'defenestrate']
2 for x in a:
3     print x, len(x)
```

Try this out ...

```
1 >>> a = ['cat', 'window', 'defenestrate']
2 >>> for x in a:
3 ...     print x, len(x)
4 ...
5 cat 3
6 window 6
7 defenestrate 12
```

BOSTON UNIVERSITY

if and for

You can control what happens inside a `for` loop using `if elif else` statements:

```
1 for x in a:
2     if len(x) > 6:
3         print x + " is important!"
4     else:
5         print x + ", so what?"
```

```
1 cat, so what?
2 window, so what?
3 defenestrate is important!
```

if and for

Don't change the list in the loop though!!!!. Make a copy:

```
1 >>> for x in a[:]: # make a slice copy of the entire list
2 ...     if len(x) > 6: a.insert(0, x)
3 ...
4 >>> a
5 ['defenestrate', 'cat', 'window', 'defenestrate']
```

Nested Loops

You can of course have nested `for` loops:

```
1 print "Ice cream Menu:"
2 for style in ['dish', 'cone']:
3     for flavor in ['vanilla', 'chocolate', 'strawberry']:
4         print style + " " + flavor
```

```
1 >>> print "Ice cream Menu:"
2 Ice cream Menu:
3 >>> for style in ['dish', 'cone']:
4 ...     for flavor in ['vanilla', 'chocolate', 'strawberry']:
5 ...         print style + " " + flavor
6 ...
7 dish vanilla
8 dish chocolate
9 dish strawberry
10 cone vanilla
11 cone chocolate
12 cone strawberry
```

BOSTON UNIVERSITY

range()

In other languages (I'm thinking of C), `for` loops involve iteration through a list of values. In Python this is accomplished by using the `range()` function.

```
1 >>> range(10)
2 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`range()` can actually take three arguments. Try:

```
1 range(5, 10)
2 range(0, 10, 3)
3 range(-10, -40, -70)
```

What does it mean to supply only one argument?
What does it mean to supply only two arguments?
What does it mean to supply all three arguments?

BOSTON UNIVERSITY

range()

```
1 >>> range(5, 10)
2 [5, 6, 7, 8, 9]
3 >>> range(0, 10, 3)
4 [0, 3, 6, 9]
5 >>> range(-10, -100, -30)
6 [-10, -40, -70]
```

range(10) - returns a list of numbers from 0 (default) to 9 (10-1) in steps of 1 (default)

range(5, 10) - returns a list of numbers from 5 to 9 (10-1) in steps of 1 (default)

range(0, 10, 3) - returns a list of numbers from 0 to 9 (10-1) in steps of 3

BOSTON UNIVERSITY

range()

One way to get both the element and index of a list is to loop in a more traditional fashion:

```
1 >>> a = ['Mary', 'had', 'a', 'little', 'lamb']
2 >>> for i in range(len(a)):
3 ...     print i, a[i]
4 ...
5 0 Mary
6 1 had
7 2 a
8 3 little
9 4 lamb
```

More dot-product

Look at `practice\dot_product2.py` and try the assignment.

BOSTON UNIVERSITY

File Access

I have some data in `examples\life_expectancies_usa.txt`
go there (`% cd examples`) and try this out:

```
3 data = open("life_expectancies_usa.txt", "r")
4 type(data)
5 for line in data:
6     print line
```

`open(name, mode)` - `mode` can be

- read ('r')
- write ('w')
- append('a')
- Adding 'b' to mode makes it binary, 'rb' is read binary

File Access

- `file.read()` - everything as a string
- `file.readline()` - read only next line
- `file.readlines()` - read all lines as a list
- `file.write()`: don't forget the newline '`\n`'

While Loops

Python also provides a `while` looping method. While a condition is `True`, the **code block** is *repeatedly* executed:

```
1 count = 0
2 while (count < 4):
3     print 'The count is:', count
4     count = count + 1
```

```
1 >>> count = 0
2 >>> while (count < 4):
3 ...     print 'The count is:', count
4 ...     count = count + 1
5 ...
6 The count is: 0
7 The count is: 1
8 The count is: 2
9 The count is: 3
```

Infinite Loops

You need to be careful because you can end up in *infinite loops* when using `while` blocks:

```
1 arrived = False
2 while not arrived:
3     print "Are we there yet?"
```

This is a never ending program, 1) how does it end?
2) why do this?
Answers:
1) Ctrl-C (`KeyboardInterrupt`)
2) You are waiting for an unpredictable event to occur.

If and While Loops

Fine control of `while` blocks can also come from `if`, etc.

```
1  arrived = False
2  count = 0
3  while not arrived:
4      print "Back-seat: Are we there yet?"
5      count = count + 1
6      if count > 100:
7          print "Front-seat: I've had it up to here with you kids!"
8      elif count > 50:
9          print "Front-seat: Yes"
10     else:
11         print "Front-seat: Soon.
```

But what if you want to stop the loop depending on an unpredictable condition?

break

`break` is the keyword you're looking for. It causes a loop, of any kind, to stop happening:

```
1 count = 0
2 while True: #infinite loop!!
3     print "Back-seat: Are we there yet?"
4     count = count + 1
5     if count > 10:
6         print "Front-seat: Yes!"
7         break
8     else:
9         print "Front-seat: Nope!"
```

Getting info from user

Python provides a method, `raw_input()`, that is used to prompt the user to respond through the console. The only argument is the value of the prompt:

```
1 name = raw_input("What's your name? ")
2 print "The user's name is " + name
```

This only works in Python scripts.
Also keep in mind, the returned value is a string.
Use `int()` or `float()` to convert this string to an interger or a float.

More dot-product

Look at `practice\dot_product3.py` and try the assignment.

BOSTON UNIVERSITY

Dict - Purpose

- A python `dict` type is an *associative array*
- Unlike lists, which are indexed by integers, dictionaries are indexed by *keys*, which can be any *immutable* type.
- Numbers and Strings are immutable, so they can be used as keys.
- Expert tip: What if I need 'variable' variable names? ... Use a dictionary.

BOSTON UNIVERSITY

Initialization

- While lists are defined using [and], dictionaries are defined using curly brackets: { and }.
- Each key, value pair is connected by a semi-colon: `'key' : 'value'`
- And a dictionary is a comma-separated list of these key, value pairs

```
1 >>> tel = {'jack': 4098, 'sape': 4139}
2 >>> tel
3 {'sape': 4139, 'jack': 4098}
4 >>> type(tel)
5 <type 'dict'>
```

Accessing Elements

- You access elements of a dictionary in much the same fashion as with a list: `my_dict['key']`
- You can `get('key', 'default')` an element, which will safely return a value if the 'key' does not exist.

```
1 >>> tel['jack']
2 4098
3 >>> print "sape's telephone number is", tel['sape']
4 sape's telephone number is 4139
5 >>> print "guido's telephone number is", tel.get('guido', 4127)
6 guido's telephone number is 4127
7 >>> tel
8 {'sape': 4139, 'jack': 4098}
```

BOSTON UNIVERSITY

Adding Elements

- You add elements of a dictionary in much the same fashion as with a list: `my_dict['key'] = 'value'`
- You can also `setdefault['key', 'value']`, which acts much like `get()`, but also saves the pair.

```
1 >>> tel['guido'] = 4127
2 >>> tel
3 {'sape': 4139, 'guido': 4127, 'jack': 4098}
4 >>> print "john's telephone number is", tel.setdefault('john', 411
5 john's telephone number is 4118
6 >>> tel
7 {'sape': 4139, 'john': 4118, 'jack': 4098, 'guido': 4127}
```

BOSTON UNIVERSITY

Changing Elements

- You change elements of a dictionary in much the same fashion as with a list: `my_dict['key'] = 'value'`
- `del` allows you to delete a key, value pair.

```
1 >>> tel
2 {'sape': 4139, 'john': 4118, 'jack': 4098, 'guido': 4127}
3 >>> tel['sape'] = 4039
4 >>> del tel['jack']
5 >>> tel
6 {'sape': 4039, 'john': 4118, 'guido': 4127}
```

`keys()` and `values()`

- `keys()` returns a *list* of the dict's keys
- `values()` returns a *list* of the dict's values

```
1 >>> print "There are", len(tel.values()),"telephone numbers"
2 There are 3 telephone numbers
3 >>> for key in tel.keys():
4 ...     print key,"has telephone number",tel[key]
5 ...
6 sape has telephone number 4039
7 john has telephone number 4118
8 guido has telephone number 4127
```

How is the order of keys determined?

Modules - Purpose

- Like scripts, *modules* provide a way to store your work from session to session.
- Each module is contained within a file, so every script you write is in fact a module, too.
- Modules can be used like a *namespace*.
- This means they can be used to isolate your code from other code that has the same name.
- What if you wanted to make your own `len` function? There would be a conflict with the default `len`. Use a module.

BOSTON UNIVERSITY

builtins

- Python already comes with many standard methods, `len`, `range`, etc.
- These are associated with a module called `__builtin__`.
- The contents of this module are automatically loaded and accessible when you start a session or a script.
- The point is that importing code you have elsewhere is consistent with how the basics of Python works.

BOSTON UNIVERSITY

imports

- If you want code that is not standard, you must `import` it:
  `import my_module` accesses `my_module.py`.
- Once imported, you access the modules elements using a period:
  `my_module.some_func` or `my_module.some_var`
- You can rename it to something else using `as`:
  `import my_module as mymod`
- You can get just one component by using `from`:
  `from my_module import some_func`. Now just use `some_func`.
- In a pinch you can get everything using `*`

Use `*` with caution, it can lead to bugs.

random

- `random` is a module that has several functions to perform (*pseudo*) random operations.
- In random.py is some functions we are going to use for a little project
- `randint` returns a random integer from `a` to `b`
- `choice` returns a random value from a list.

BOSTON UNIVERSITY

random.randint

```
1  >>> import random
2  >>> help(random.randint)
3  Help on method randint in module random:
4
5  randint(self, a, b) method of random.Random instance
6      Return random integer in range [a, b], including both end poin
7
8  >>> random.randint(0,9)
9  1
10 >>> random.randint(0,9)
11 4
12 >>> random.randint(0,9)
13 4
14 >>> random.randint(0,9)
15 8
```

random.choice

```
>>> import random
>>> help(random.choice)
Help on method choice in module random:

choice(self, seq) method of random.Random instance
    Choose a random element from a non-empty sequence.

>>> lucky = ['Tim', 'Tom', 'Ted', 'Tony']
>>> type(lucky)
<type 'list'>
>>> random.choice(lucky)
'Tony'
>>> random.choice(lucky)
'Tony'
>>> random.choice(lucky)
'Ted'
```

An Example

Check out `examples\state_capitals.py`.

1. Run it.
2. Open it and read how it works.
3. Ask questions if you are lost...

BOSTON UNIVERSITY

References

- This Tutorial was based on a tutorial presented by the Boston Python Group:

  `https://openhatch.org/wiki/Boston_Python_Workshop_6`

  Look at "Saturday projects" for your next step.
- Follow the official Python tutorial for a full look at all the basic features:

  `http://docs.python.org/tutorial/index.html`
- Code, code, code; codingbat.com, udacity.com, etc.
- Ask for help: yannpaul@bu.edu

BOSTON UNIVERSITY