

# Introduction to Python 1

Chang Y. Chung

Office of Population Research

01/14/2014

## Why Python

- ▶ Popular
- ▶ Easy to learn and use
- ▶ Open-source
- ▶ General-Purpose
- ▶ Multi-paradigm (procedural, object-oriented, functional)
- ▶ Hettinger, “What makes Python Awesome?”[2]  
<http://tinyurl.com/mn4d4er>
- ▶ Did I mention *popular*?

1 / 46

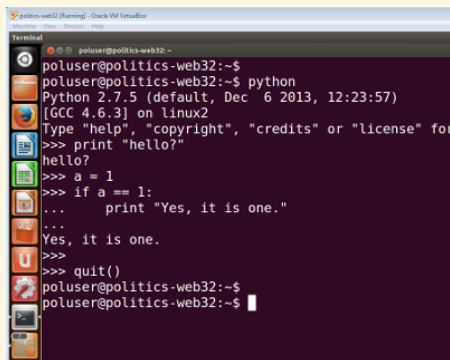
## Organization

- ▶ First two hours for learning basics of Python language.
- ▶ Third hour for a guided tour of Python “ecosystem”.

2 / 46

## Running Python Read-Eval-Print Loop

- ▶ For those who are using UNIX-like systems, including the Ubuntu image running on a Virtual Box.
- ▶ Type "python" at the terminal shell prompt.

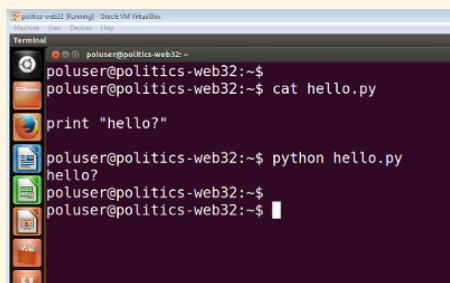


```
poluser@politics-web32:~$ python
Python 2.7.5 (default, Dec 6 2013, 12:23:57)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license()" for more
>>> print "hello?"
hello?
>>> a = 1
>>> if a == 1:
...     print "Yes, it is one."
...
Yes, it is one.
>>> quit()
poluser@politics-web32:~$
```

3 / 46

## Running a Python Script File (.py)

- ▶ Type "python" followed by the script file name.



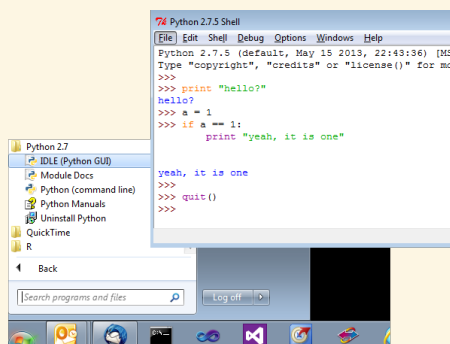
```
poluser@politics-web32:~$ cat hello.py
print "hello?"

poluser@politics-web32:~$ python hello.py
hello?
poluser@politics-web32:~$
```

4 / 46

## Running Python REPL on Windows

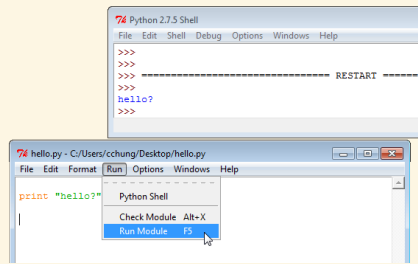
- ▶ For those who are using Windows system with Python (and IDLE) installed.
- ▶ Run the IDLE application.



5 / 46

## Running a Python Script File (.py) on Windows

- ▶ File > New (or OPEN) brings up a script window.
- ▶ Run > Run Module (Output will show in IDLE shell).



6 / 46

## Conceptual Hierarchy by Mark Lutz[3]

- ▶ Programs are composed of *modules*.
- ▶ Modules contain *statements*.

7 / 46

## Conceptual Hierarchy by Mark Lutz[3]

- ▶ Programs are composed of *modules*.
- ▶ Modules contain *statements*.
- ▶ Statements contain *expressions*.
- ▶ Expressions create and process objects.

7 / 46

## Script File (.py)

- ▶ A script file is a module.
- ▶ A script is a sequence of statements, delimited by a newline (or the end of line character).
- ▶ Python executes one statement at a time, from the top of a script file to the bottom.

8 / 46

## Script File (.py)

- ▶ A script file is a module.
- ▶ A script is a sequence of statements, delimited by a newline (or the end of line character).
- ▶ Python executes one statement at a time, from the top of a script file to the bottom.
- ▶ Execution happens in namespaces (modules, classes, functions all have their own).
- ▶ Everything is runtime (even `def`, `class`, and `import`)

8 / 46

## Executable Python Script File

- ▶ On a Unix-like system, we can change the mode of the script file and make it executable:

```
1 $chmod +x hello.py
2 $./hello.py
```

- ▶ Just add the first line with a hashbang (`#!`):

```
1 #!/usr/bin/python
2
3 # say hello to the world
4 def main():
5     print "Hello , World!"
6
7 if __name__ == "__main__":
8     main()
```

9 / 46

## Comments

- Comments start with a hash (#) and end with a newline.

```
1  # this whole line is a comment
2
3  # add 10 integers.
4  total = 0
5  for i in range(10): # i goes 0, 1, 2, ..., 9
6      total += i      # shortcut for total = total + i
7
8  print "total=", total
9  # total= 45
```

10/46

## Variables

- Variables are created when first assigned a value.

```
1  my_var = 3
2  answer_to_everything = 42
3
4  # also works are:
5  x = y = 0
6  a, b, c = 1, 2, 3
```

- Variable names start with a letter or an underscore(\_) and can have letters, underscores, or digits.
- Variable names are case-sensitive.

11/46

## Assignment Semantics According to David Godger[1]

- Variables in many *other languages* are a container that stores a value.

```
1  int a = 1;
```



12/46

## Assignment Semantics According to David Godger[1]

- Variables in many *other languages* are a container that stores a value.

```
1 int a = 1;
```



- In Python, an assignment creates an object, and labels it with the variable.

```
1 a = 1
```



12/46

## Assignment Semantics According to David Godger[1]

- Variables in many *other languages* are a container that stores a value.

```
1 int a = 1;
```



- In Python, an assignment creates an object, and labels it with the variable.

```
1 a = 1
```



- If you assign another value to *a*, then the variable labels the new value (2).

```
1 a = 2
```



12/46

## Assignment Semantics According to David Godger[1]

- Variables in many *other languages* are a container that stores a value.

```
1 int a = 1;
```



- In Python, an assignment creates an object, and labels it with the variable.

```
1 a = 1
```



- If you assign another value to *a*, then the variable labels the new value (2).

```
1 a = 2
```



- This is what happens if you assign *a* to a new variable *b*:

```
1 b = a
```



12/46

## Numbers

### ► Integers

```
1 x = 0
2 age = 20
3 size_of_household = 5
4
5 print type(age)
6 # <type 'int'>
7
8 # can handle arbitrarily large numbers
9 huge = 10 ** 100 + 1
```

### ► Floating-point Numbers

```
1 g = 0.1
2 f = 6.67384
3
4 velocity = 1. # it is the dot (.) that makes it a float
5 print velocity
6 # 1.0
7 type(velocity)
8 # <type 'float'>
```

13 / 46

## Numeric Expressions

### ► Most of the arithmetic operators behave as expected.

```
1 a = 10
2 b = 20
3 print a - (b ** 2) + 23
4 # -367
5
6 x = 2.0
7 print x / 0.1
8 # 20.0
```

### ► Watch out for integer divisions. In Python 2, it *truncates down* to an integer.

```
1 print 10 / 3
2 # 3 (in Python 2)      3.3333333333333335 (in Python 3)
3 print -10 / 3
4 # -4 (in Python 2)     -3.3333333333333335 (in Python 3)
5
6 # a solution: use floating point numbers
7 print 10.0 / 3.0
8 # 3.3333333333333335 (in both Python 2 and 3)
```

14 / 46

## String Literals

### ► A string is a sequence of characters.

```
1 # either double (") or single(') quotes for creating string literals.
2 name = "Changarilla Dingdong"
3 file_name = 'workshop.tex'
4
5 # triple-quoted string
6 starwars = """
7 A long time ago is a galaxy far, far away...
8
9 It is a period of civil war. Rebel
10 spaceships, striking from a hidden
11 base, have won their first victory
12 ...
13
14 What is the last character of this string?
15 """
16
17 last_char = starwars[-1]
18 print ord(last_char), ord("\n")
19 # 10 10
```

15 / 46

## Working with strings

- Strings are immutable.

```
1 s = "abcde"
2 s[0] = "x" # trying to change the first char to "x"
3 # TypeError: 'str' object does not support item assignment
4
5 t = "x" + s[1:] # creating a new string
6 print t
7 # xbcde
```

- Many functions and methods are available.

```
1 s = "abcde"
2 print s + s # concatenation
3 # abcdeabcde
4
5 print len(s)
6 # 5
7
8 print s.find("c") # index is 0-based. returns -1 if not found
9 # 2
```

16/46

## String Manipulation

- A few more string methods.

```
1 s = "abcde"
2 print s.upper(), "XYZ".lower()
3 # ABCDE, xyz
4
5 print "  xxx  yy  ".strip()
6 # xxx yy
7
8 print "a,bb,ccc".split(",")
9 # ['a', 'bb', 'ccc']
```

17/46

## String Manipulation

- A few more string methods.

```
1 s = "abcde"
2 print s.upper(), "XYZ".lower()
3 # ABCDE, xyz
4
5 print "  xxx  yy  ".strip()
6 # xxx yy
7
8 print "a,bb,ccc".split(",")
9 # ['a', 'bb', 'ccc']
```

- What are "methods"?

17/46



## String Manipulation

### ► A few more string methods.

```
1 s = "abcde"
2 print s.upper(), "XYZ".lower()
3 # ABCDE, xyz
4
5 print "   xxx yy   ".strip()
6 # xxx yy
7
8 print "a,bb,ccc".split(",")
9 # ['a', 'bb', 'ccc']
```

### ► What are "methods"?

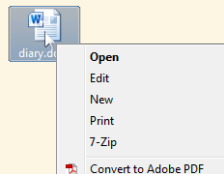
- ▷ Functions which are a member of a type (or class).

17 / 46

## String Manipulation

### ► A few more string methods.

```
1 s = "abcde"
2 print s.upper(), "XYZ".lower()
3 # ABCDE, xyz
4
5 print "   xxx yy   ".strip()
6 # xxx yy
7
8 print "a,bb,ccc".split(",")
9 # ['a', 'bb', 'ccc']
```



### ► What are "methods"?

- ▷ Functions which are a member of a type (or class).
- ▷ `int`, `str`, `Word Document` are types (or classes).
- ▷ `2`, `"abcde"`, `diary.docx` are an *instance* (or *object*) of the respective type.
- ▷ Types have members: properties (data) and methods (functions).

17 / 46

## Two Ways to Format

### ► `format()` method.

```
1 print "The answer is {}".format(21)
2 # The answer is 21
3 print "The real answer is {:.4f}".format(21.2345678)
4 # The answer is 21.2346
```

### ► Formatting operator.

```
1 print "The answer is %d" % 21
2 # The answer is 21
3 print "The real answer is %.4f" % 21.2345678
4 # The answer is 21.2346
```

18 / 46

## Raw Strings

- ▶ Within a string literal, escape sequences start with a backslash (\)

```
1 a_string = 'It\'s a great day\nto learn \\Python\\.\n 1\t 2\t 3'
2 print a_string
3 # It's a great day
4 # to learn \Python\
5 # 1      2      3
```

- ▶ A raw string literal starts with the prefix r. In a raw string, the backslash (\) is not special.

```
1 import re
2
3 # raw strings are great for writing regular expression patterns.
4 p = re.compile(r"\d\d\d-\d\d\d\d")
5 m = p.match('123-4567')
6 if m is not None:
7     print m.group()
8 # 123-4567
```

19 / 46

## Unicode Strings

- ▶ You can create Unicode strings using the u prefix and "\u" escape sequences.

```
1 a_string = u"Euro \u20AC" # \u followed by 16-bit hex value xxxx
2 print a_string, len(a_string)
3 # Euro € 6
```

20 / 46

## Unicode Strings

- ▶ You can create Unicode strings using the u prefix and "\u" escape sequences.

```
1 a_string = u"Euro \u20AC" # \u followed by 16-bit hex value xxxx
2 print a_string, len(a_string)
3 # Euro € 6
```

- ▶ Unicode strings are sequences of *code points*.
- ▶ Code points are numbers, each representing a “character”. e.g., U+0061 is ‘Latin small letter a’.
- ▶ Unicode text strings are *encoded* into bytes. UTF-8 is one of many Unicode encodings, using one to four bytes to store a Unicode “character”.
- ▶ Once you have Unicode strings in Python, all the string functions and properties work as expected.

20 / 46

## Best Practices According to Thomas Wouters[5]

- ▶ Never mix unicode and bytecode [i.e. ordinary] strings.
- ▶ Decode bytecode strings on input.
- ▶ Encode unicode strings on output.
- ▶ Try automatic conversion (`codecs.open()`)
- ▶ Pay attention to exceptions, `UnicodeDecodeError`
- ▶ An example

```
1 ustr = u"Euro \u20AC" # Euro €
2
3 # Python's default encoding codec is 'ascii'
4 ustr.encode()
5 # UnicodeEncodeError: 'ascii' codec can't encode character 'u\u20ac'
6 utf_8 = ustr.encode("UTF-8") # encoding to UTF-8 works fine
7 # this takes 8 bytes. five one-byte's and one three-byte
8
9 # now we want to decode to ascii, ignoring non-ascii chars
10 print utf_8.decode("ascii", "ignore")
11 # Euro (no euro symbol character)
```

21 / 46

## None

- ▶ Is a place-holder, like NULL in other languages.

```
1 x = None
```

- ▶ Is a universal object, i.e., there is only one None.

```
1 print None is None
2 # True
```

- ▶ Is evaluated as False.

```
1 x = None
2 if x:
3     print "this will never print"
```

- ▶ Is, however, distinct from others which are False.

```
1 print None is 0, None is False, None is []
2 # False, False, False
```

22 / 46

## Core Data Types

- ▶ Basic core data types: int, float, and str.
- ▶ "Python is dynamically, but *strongly* typed."

```
1 n = 1.0
2 print n + "99"
3 # TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- ▶ Use `int()` or `float()` to go from str to numeric

```
1 print n + float("99")
2 # 100.0
```

- ▶ `str()` returns the string representation of the given object.

```
1 n = 1.0
2 print str(n) + "99"
3 # 1.099
```

23 / 46

## Operators

### ► Python supports following types of operators:

- ▷ Arithmetic (+ - \* / % \*\* //)
- ▷ Comparison (== != > < >= <=)
- ▷ Assignment (= += -= \*= /= %= \*\*= //=)
- ▷ Logical (and or not)
- ▷ Bitwise (& | ^ ~ << >>)
- ▷ Membership (in not in)
- ▷ Identity (is is not)

24 / 46

## A Few Surprises

### ► Power operator (\*\*) binds more tightly than unary operators on the left.

```
1 a = -2**2
2 print a
3 # -4
4 # solution: parenthesize the base
5 print (-2)**2
6 # 4
```

### ► Comparisons can be chained.

```
1 x < y <= z          # y is evaluated only once here
```

is equivalent to

```
1 x < y and y <= z
```

### ► Logical operators (and, or) short-circuit evaluation and return an *operand*.

```
1 print 3 or 2
2 # 3
```

25 / 46

## Quiz

### ► Demographic and Health Surveys (DHS) Century Month Code (CMC)[4, p.5] provides an easy way working with year and month.

The CMC is an integer representing a month, taking the value of 1 in January 1900, 2 in February 1900, ..., 13 in January 1901, etc. The CMC in February 2011 is 1333.

What is the CMC for this month, i.e., January, 2014?

26 / 46

## Quiz

- Demographic and Health Surveys (DHS) Century Month Code (CMC)[4, p.5] provides an easy way working with year and month.

The CMC is an integer representing a month, taking the value of 1 in January 1900, 2 in February 1900, ..., 13 in January 1901, etc. The CMC in February 2011 is 1333.

What is the CMC for this month, i.e., January, 2014?

- An answer:

```
1 year = 2014
2 month = 1
3 print "CMC", (year - 1900) * 12 + month
4 # CMC 1369
```

26 / 46

## Quiz

- Demographic and Health Surveys (DHS) Century Month Code (CMC)[4, p.5] provides an easy way working with year and month.

The CMC is an integer representing a month, taking the value of 1 in January 1900, 2 in February 1900, ..., 13 in January 1901, etc. The CMC in February 2011 is 1333.

What is the CMC for this month, i.e., January, 2014?

- An answer:

```
1 year = 2014
2 month = 1
3 print "CMC", (year - 1900) * 12 + month
4 # CMC 1369
```

- What is the month (and year) of CMC 1000?

26 / 46

## Quiz

- According to U.S. National Debt Clock, the current outstanding public debt, as of a day last month, was a big number:

```
1 debt = 17234623718339.96
```

Count how many times the digit 3 appears in the number. (Hint: create a string variable and use the `count()` method of the string type.)

27 / 46

## Quiz

- ▶ According to U.S. National Debt Clock, the current outstanding public debt, as of a day last month, was a big number:

```
1 debt = 17234623718339.96
```

Count how many times the digit 3 appears in the number.  
(Hint: create a string variable and use the `count()` method of the string type.)

- ▶ An answer:

```
1 sdebt = "17234623718339.96"
2 print sdebt.count("3")
3 # 4
```

27 / 46

## Quiz

- ▶ According to U.S. National Debt Clock, the current outstanding public debt, as of a day last month, was a big number:

```
1 debt = 17234623718339.96
```

Count how many times the digit 3 appears in the number.  
(Hint: create a string variable and use the `count()` method of the string type.)

- ▶ An answer:

```
1 sdebt = "17234623718339.96"
2 print sdebt.count("3")
3 # 4
```

- ▶ (tricky) It feels rather silly to rewrite the value as a string. Can you think of a way to *convert* the number into a string?

27 / 46

## Flow Control

- ▶ Conditional Execution (`if`)

- ▶ Iteration

- ▷ `while` loop
- ▷ `for` loop

28 / 46

## IF

- ▶ IF statement is used for conditional execution.

```
1 if x > 0:
2     print "x is positive"
3 else:
4     print "x is zero or negative"
```

- ▶ Only one suite (block of statements) under a `True` conditional expression is executed.

```
1 me = "rock"
2 wins = 0
3
4 if you == "paper":
5     print "You win!"
6 elif you == "scissors":
7     print "I win!"
8     wins += 1
9 else:
10    print "draw"
```

29 / 46

## Compound statements

- ▶ `if`, `while`, and `for` are *compound* statements, which have one or more *clauses*. A *clause*, in turn, consists of a *header* that ends with a colon (`:`) and a *suite*.
- ▶ A suite, a block of statements, is identified by *indentation*.

```
1 a = 1
2 if a > 0:
3     desc = "a is positive"      # these two lines form a suite
4     print a, desc              #
5                                # (blank line ignored)
6 print "done"                  # This line being started "dedented"
7                                # signals the end of the block
8
9     if a > 0:                  # indentation error
10    desc = "a is positive"      # indentation error
11
12 if a > 0:                      # OK
13     desc = "a is positive"    # OK
14     print a, desc             # OK
15 else:                          # OK
16     print a                   # OK
```

30 / 46

## Compound statements

- ▶ The amount of indentation does not matter (as long as the same within a level). Four spaces (per level) and no tabs are the convention.
- ▶ Use editor's python mode, which prevents/converts `<TAB>` to (four) spaces.
- ▶ Why indentation? A good answer at: <http://tinyurl.com/kxv9vts>.
- ▶ For an empty suite, use `pass` statement, which does nothing.

31 / 46

## WHILE

- Repeats a block of statements as long as the condition remains `True`.

```
1 total = 0
2 n = 0
3 while n < 10:
4     print n, total
5     n += 1
6     total += n
7 # 0 0
8 # 1 1
9 # 2 3
10 # ...
11 # 9 45
```

- `break` terminates the loop immediately.
- `continue` skips the remainder of the block and goes back to the test condition at the top.

32 / 46

## FOR

- `for` is used to iterate over a sequence.

```
1 days = ["Sunday", "Monday", "Tuesday", "Wednesday",
2         "Thursday", "Friday", "Saturday"]
3
4 for day in days:
5     if day == "Friday":
6         print "I am outta here."
7         break
8     print "Happy" + " " + day + "!"
9
10 # Happy Sunday!
11 # Happy Monday!
12 # ...
13 # Happy Thursday!
14 # I am outta here.
```

33 / 46

## FOR

- Another example.

```
1 numbers = range(5)
2 print numbers
3 # [0, 1, 2, 3, 4]
4
5 for n in numbers:
6     print n,
7     if n % 2 == 0:
8         print "is even"
9     else:
10        print "is odd"
11
12 # 0 is even
13 # 1 is odd
14 # 2 is even
15 # 3 is odd
16 # 4 is even
```

34 / 46



## File I/O

- ▶ The built-in function, `open()`, returns a `file` type object, unless there is an error opening the file.

```
1 in_file = open("yourfile.txt", "r") # for reading
2 out_file = open("myfile.txt", "w") # for writing
```

- ▶ Once we get the `file` type object, then use its methods.

```
1 # read all the contents from the input file
2 content = in_file.read()
3
4 # write out a line to the output file
5 out_file.write("hello?\n")
6
7 # close it when done
8 in_file.close()
9 out_file.close()
```

35 / 46

## Reading a file one line at a time

- ▶ `with` ensures that the file is closed when done.

```
1 with open("lorem.txt", "r") as f:
2     for line in f:
3         print line
```

- ▶ Another example.

```
1 # creating a file and writing three lines
2 with open("small.txt", "w") as f:
3     f.write("first\n")
4     f.write("second\n")
5     f.write("third")
6
7 with open("small.txt", "r") as f:
8     for line in f:
9         print line    # line includes the newline char
10
11 # first
12 #
13 # second
14 #
15 # third
```

36 / 46

## Defining and Calling a Function

- ▶ Defined with `def` and called by name followed by `()`.

```
1 def the_answer():
2     return 42
3
4 print the_answer()
5 # 42
```

- ▶ Argument(s) can be passed.

```
1 def shout(what):
2     print what.upper() + "!"
3
4 shout("hello")
5 # HELLO!
```

- ▶ If the function returns nothing, then it returns `None`.

```
1 r = shout("hi")
2 # Hi!
3 print r
4 # None
```

37 / 46

## Another example

### ► CMC again

```
1 def cmc(year, month):
2     ''' returns DHS Century Month Code ''' # doc string
3     if year < 1900 or year > 2099:
4         print "year out of range"
5         return
6     if month < 1 or month > 12:
7         print "month out of range"
8         return
9     value = (year - 1900) * 12 + month
10    return value
11
12 print cmc(2014, 1)
13 # 1369
14 print cmc(2014, 15)
15 # month out of range
16 # None
```

38 / 46

## Local and Global Variables

### ► Within your function:

- ▷ A new variable is *local*, and independent of the global var with the same name, if any.

```
1 x = 1 # global (or module)
2
3 def my_func():
4     x = 2 # local
5     my_func()
6     print x # 1
```

- ▷ Both local and global variables can be read.
- ▷ Global variables can be written to once *declared* so.

```
1 x = 1
2
3 def my_func():
4     global x
5     x = 2 # global
6     my_func()
7     print x # 2
```

39 / 46

## Quiz

- Write a function that returns Body Mass Index (BMI) of an adult given weight in kilograms and height in meters.  
(Hint: BMI = weight(kg) / (height(m) squared).  
For instance, if a person is 70kg and 1.80m, then BMI is about 21.6.)

40 / 46

## Quiz

- ▶ Write a function that returns Body Mass Index (BMI) of an adult given weight in kilograms and height in meters. (Hint:  $BMI = \text{weight(kg)} / (\text{height(m)}^2)$ . For instance, if a person is 70kg and 1.80m, then BMI is about 21.6.)
- ▶ An answer.

```
1 def bmi(kg, m):  
2     return float(kg) / (m ** 2)
```

40 / 46

## Quiz

- ▶ Write a function that returns Body Mass Index (BMI) of an adult given weight in kilograms and height in meters. (Hint:  $BMI = \text{weight(kg)} / (\text{height(m)}^2)$ . For instance, if a person is 70kg and 1.80m, then BMI is about 21.6.)
  - ▶ An answer.
- ```
1 def bmi(kg, m):  
2     return float(kg) / (m ** 2)
```
- ▶ Re-write the bmi function so that it accepts height in feet and inches, and the weight in pounds. (Hint. Make pound, foot, and inch arguments. Convert them into local variables, kg and m, before calculating bmi to return.)

40 / 46

## Importing a module

- ▶ `import` reads in a module, runs it (top-to-bottom) to create the module object.
- ▶ Via the module object, you get access to its variables, functions, classes, ...
- ▶ We've already seen an example of importing a standard regular expression module:

```
1 import re  
2  
3 # compile() is a function defined within the imported re module.  
4  
5 p = re.compile(r"\d\d\d-\d\d\d\d")  
6 m = p.match('123-4567')  
7 if m is not None:  
8     print m.group()  
9 # 123-4567
```

41 / 46

## Another example

- There are many standard modules that come already installed, and be ready to be imported.

```
1 import math
2
3 s = math.sqrt(4.0)
4 print "4.0 squared is {:.2f}".format(s)
5 # 4.0 squared is 2.00
```

- You can selectively import as well.

```
1 from math import sqrt # import sqrt() alone
2
3 print sqrt(9.0)        # no "math."
4 # 3.0
```

- You can import your own Python script file (.py) the same way. The default import path includes the current working directory and its sub-directories.

```
1 import hello # suppose that hello.py defines a main() function
2
3 hello.main()
4 # "hello, World!"
```

42 / 46

## Quiz

- Write a function such that, given a BMI value, returns the BMI category as a string. Recall the categories are:

|               |                                |
|---------------|--------------------------------|
| Underweight   | less than 18.5                 |
| Normal weight | 18.5 upto but not including 25 |
| Overweight    | 25 upto but not including 30   |
| Obesity       | 30 or greater                  |

For instance, the function should return a string "Normal weight", when it is called with an argument of, say 20. (Hint: use conditional statements i.e., if ... elif ...)

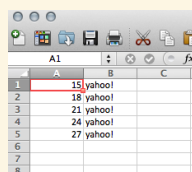
- Print out a BMI table showing several lines of a pair: a BMI value and its category. BMI value may start at 15 and go up by 3 up to 36. (Hint: use a loop)

43 / 46

## Quiz (cont.)

- Create a comma-separated values (.csv) file of the BMI table. (Hint: You may start with below, and modify it as necessary.)

```
1 import csv
2
3 with open("test.csv", "wb") as f:
4     my_writer = csv.writer(f)
5     i = 15
6     while i < 30:
7         my_writer.writerow([i, "yahoo!"])
8         i += 3
```



|   | A1 | B1     | C1 |
|---|----|--------|----|
| 1 | 15 | yahoo! |    |
| 2 | 18 | yahoo! |    |
| 3 | 21 | yahoo! |    |
| 4 | 24 | yahoo! |    |
| 5 | 27 | yahoo! |    |
| 6 |    |        |    |
| 7 |    |        |    |
| 8 |    |        |    |

44 / 46

## Summary

- ▶ Using Python interactively or by running a script.
- ▶ Comments.
- ▶ Variables and assignment semantics.
- ▶ Core data types (int, float, str, None).
- ▶ Operators.
- ▶ Conditionals and Looping.
- ▶ Defining and using functions.
- ▶ Basic File I/O.
- ▶ Importing a module.

45 / 46

## References

-  Godger, D.  
Code like a pythonista: Idiomatic python.  
<http://tinyurl.com/2cv9kg>.
-  Hettinger, R.  
What makes Python Awesome?  
<http://tinyurl.com/mn44er>.
-  Lutz, M.  
*Learning Python*, fifth ed.  
O'Reilly Media, Sebastopol, CA, 2013.
-  MEASURE DHS Plus.  
*Description of the Demographic and Health Surveys Individual Recode Data File*, version 1.0 ed., March 2008.
-  Wouters, T.  
Advanced python: (or understanding python) google tech talks. feb 21, 2007.  
<http://www.youtube.com/watch?v=u0zdG3lwcB4>.

46 / 46