# STSCI 4060

## Lecture File 3

**Xiaolong Yang**
**(xy44@cornell.edu)**

Xiaolong Yang © 2019

# Python Statements

A statement is a section of Python code that represents a command or an action, which is actually an instruction that the Python interpreter can execute.

- **Simple statements** (those that do *not* contain a nested block)
  - print statement (e.g., print 'My name is %s.' % name)
  - assignment statement (e.g., a=[1,2,3])
  - import statement
  - assert statement
- **Compound statements** (for language control)
  - if: statement
  - for: statement
  - while: statement
  - try: … except: statements:

Xiaolong Yang © 2019

# Simple Python Statements

- import statement
- assert statement

Xiaolong Yang © 2019

# The *import* Statement

- The import statement makes a module and its contents available for use.
- The import statement *evaluates* the code in a module, but only the first time when any given module is imported into an application.
- The import statement has many forms:
  - import module_name
  - import module_name1, module_name2, …
  - from module_name import x
  - from module_name import x, y, …
  - from module_name import *
  - import module_name as new_ module_name
  - from module_name import x as new_ x

Xiaolong Yang © 2019

# The *import* Statement: Examples

- ***import* module_name**

  Import all the functions from the module but have to prefix the module name every time you use a function.

```
>>> import math
>>> math.pi
 3.141592653589793
>>> pi
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    pi
NameError: name 'pi' is not defined
>>> math.sqrt(2)
1.4142135623730951
>>> sqrt(2)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    sqrt(2)
NameError: name 'sqrt' is not defined
```

# The *import* Statement: Examples

- **from module_name *import* x**

  Import only one function x from the module but do not prefix the module name when you use the function.

```
>>> from random import randrange
>>> randrange(10, 100)
10
>>> randrange(10, 100, 2)
26
>>> random.randrang(10, 100)   #prefixing the function
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    random.randrang(10, 100)
NameError: name 'random' is not defined
>>> randint(1, 10)      #using another function in the random module
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    randint(1, 10)
NameError: name 'randint' is not defined
```

Xiaolong Yang © 2019

# The *import* Statement: Examples

- **from module_name *import* x, y**
  Import only functions x and y from the module but do not prefix the module name when you use the functions.

```
>>> from random import randrange, randint, seed
>>> randrange(10, 100)
66
>>> randint(1, 10)
2
>>> seed(2)
>>> randint(1,100)
96
>>> randint(1,100)
95
>>> seed(2)
>>> randint(1,100)
96
>>> randint(1,100)
95
```

Xiaolong Yang © 2019

# The *import* Statement: Examples

- **from module_name *import* *** (wildcard import)
  Import all the attributes of the module but should use this form of the import statement with caution. It may cause confusions of what names exist in the namespace. If same named variables already exist, they are replace by the new ones just imported. However, the advantage is, again, that you do not prefix the module name when you use a function.

```
>>> from math import *
>>> pi
3.141592653589793
>>> sqrt(2)
1.4142135623730951
>>> math.pi
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    math.pi
NameError: name 'math' is not defined
>>> from random import *
>>> randint(1, 10)
5
```

# The *import* Statement: Examples

- ***import* module_name as new_ module_name**
  You can rename an imported module to a different name you prefer.

```
>>> import  math as myMath
>>> myMath.pi
3.141592653589793
>>> math.pi
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    math.pi
NameError: name 'math' is not defined
```

# The *import* Statement: Examples

- ***from module_name import x as new_ x***
  You can rename a function (x) imported from a module to a different name, new_x, which you prefer.

```
>>> from random import randrange as myRand
>>> myRand(1,1000)
787
>>> randrange(1,1000)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    randrange(1,1000)
NameError: name 'randrange' is not defined
```

Xiaolong Yang © 2019

# Create a User-defined Module for Importing

fruit_module.py

```
#This program displays fruit info and does simple accounting
def fruit(fname, edible):
    print
    print '*' * 50
    print 'Fruit name (upper case): %s.' % fname.upper()
    if edible:
        print "It's edible."
        quantity=input('How many %sS do you want? ' % fname.upper())
        price=input('What is the price? ')
        amount=quantity*price
        print "Please pay $" + str(amount) + '.'
    else:
        print "It's not edible. Probably you do not want it."
```

# Create a User-defined Module for Importing: the Output

```
>>> import fruit_module as fm
>>> fm.fruit('apple',True)

**********************************************
Fruit name (upper case): APPLE.
It's edible.
How many APPLES do you want? 15
What is the price? 1.38
Please pay $20.7.
>>> fm.fruit('spoiled apple',False)

**********************************************
Fruit name (upper case): SPOILED APPLE.
It's not edible. Probably you do not want it.
>>>
```

# Modify the module to Achieve the Following

```
>>> import fruit_module_2 as fm2
>>> fm2.fruit('apple', 1)
**************************************************
Fruit name (upper case): APPLE.
It's edible.
How many APPLES do you want? 15
What is the price? 1.38
Please pay $20.7.
>>> fm2.fruit('spoiled peach', 0)
**************************************************
Fruit name (upper case): SPOILED PEACH.
It's not edible. Do you still want to buy it? Y(es)/N(o)? Yes
Sorry, these cannot be sold any more. Goodbye!
>>> fm2.fruit('spoiled peach', 0)
**************************************************
Fruit name (upper case): SPOILED PEACH.
It's not edible. Do you still want to buy it? Y(es)/N(o)? y
Sorry, these cannot be sold any more. Goodbye!
>>> fm2.fruit('spoiled peach', 0)
**************************************************
Fruit name (upper case): SPOILED PEACH.
It's not edible. Do you still want to buy it? Y(es)/N(o)? No
Thank you. Goodbye!
>>> fm2.fruit('spoiled peach', 0)
**************************************************
Fruit name (upper case): SPOILED PEACH.
It's not edible. Do you still want to buy it? Y(es)/N(o)? n
Thank you. Goodbye!
```

Xiaolong Yang © 2019

# Further Modify the module to Achieve the Following

```
>>> import fruit_module_3 as fm3
>>> fm3.fruit('bad apple', 0)
**********************************************
Fruit name (upper case): BAD APPLE.
It's not edible. Do you still want to buy it? Y(es)/N(o)? n
Thank you. Goodbye!
>>> fm3.ifno('n')
Thank you. Goodbye!
>>> fm3.ifno('Y')
Sorry, these cannot be sold any more. Goodbye!
>>>
```

Xiaolong Yang © 2019

# More: Importing a Module with Wildcard Import

fruit_module_4.py

```
#This program displays fruit info and does simple accounting

applePrice=1.28
orangePrice=0.99

def fruit(fname, edible):
    print '*' * 50
    print 'Fruit name (upper case): %s.' % fname.upper()
    if edible:
        print "It's edible."
        answerquantity=input('How many %sS do you want? ' % fname.upper())
```

# More: Importing a Module with Wildcard Import

```
>>> applePrice=5
>>> orangePrice=3
>>> from fruit_module_4 import *
>>> applePrice
1.28
>>> orangePrice
0.99
>>> fruit('apple',1)
**************************************************
Fruit name (upper case): APPLE.
It's edible.
How many APPLES do you want? 10
Please pay $12.8.
>>> fruit('orange', 1)
**************************************************
Fruit name (upper case): ORANGE.
It's edible.
How many ORANGES do you want? 10
Please pay $9.9.
>>> fruit('peach', 1)
**************************************************
Fruit name (upper case): PEACH.
It's edible.
How many PEACHS do you want? 10
What is the price? 0.89
Please pay $8.9.
```

Xiaolong Yang © 2019

# The *assert* Statement

Use the assert statement to place error checking statements in your code. If the condition is correct, the program go silently to the next statement; otherwise, this statement raises an **AssertionError exception**. Here is an example:

```
>>> def test(arg1, arg2):
        arg1 = float(arg1)
        arg2 = arg2
        assert arg2 != 0, 'Bad division; the denominator arg2 is %s.' %(arg2)
        ratio = arg1 / arg2
        print 'ratio: ', ratio
>>> test(11, 4)
ratio: 2.75
>>> test(11,0)

Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    test(11,0)
  File "<pyshell#24>", line 5, in test
    %s.' %(arg2)
AssertionError: Bad division; the denominator arg2 is  0.
```

# Compound Python Statements

- if: statement
- for: statement
- while: statement
- try: … except:

Xiaolong Yang © 2019

# The *if* Statement

This is to formally introduce the *if* statement. It enables us to execute code (or not) depending on a condition(s).

```
if condition:
    statement_block       # condition is true
```

```
if condition:
    statement_block_1    # condition is true
else:
    statement_block_2    # condition is false
```

```
if condition1:
     statement_block_1    # condition1 is true
else:
     if condition2:
          statement_block_2  # condition2 is true
     else:
          statement_block_3 # no condition is true
```

```
if condition1:
    statement_block_1
elif condition2:
    statement_block_2
    ●
    ●
    ●
else:
    statement_block_n
```

Exactly *one* of the indented blocks is executed. It is the one corresponding to the *first* True condition, or, if all conditions are False, it is the block after the final else line.

# The *if* Statement Example

```
def letterGrade(score):
    if score >= 90:
        letter = 'A'
    else: # grade must be B, C, D or F
        if score >= 80:
            letter = 'B'
        else: # grade must be C, D or F
            if score >= 70:
                letter = 'C'
            else: # grade must be D or F
                if score >= 60:
                    letter = 'D'
                else:
                    letter = 'F'
    return letter
```

≡

```
def letterGrade(score):
    if score >= 90:
        letter = 'A'
    elif score >= 80:
        letter = 'B'
    elif score >= 70:
        letter = 'C'
    elif score >= 60:
        letter = 'D'
    else:
        letter = 'F'
    return letter
```

# The *for* Statement

The *for* statement enables us to iterate over **collections** (strings, lists, tuples, and dictionaries) so that we can repeat the same block of code once for each element of the collection or repeat the block of code for certain number of times (***for loop***).

Iterate over a list:

```
>>> numbers = [100,200,300]  # a list
>>> for item in numbers:
        print 'item: ', item
item: 100
item: 200
item: 300
```

Iterate over the keys a dictionary:

```
>>> myDict = {'cat': 'furry and cute', 'dog': 'friendly  and smart'}   # a dictionary
>>> for key in myDict.keys():
   print 'A %s is %s.' % (key, myDict[key])
A dog is friendly and smart.
A cat is furry and cute.
```

# The *for* Statement (cont'd)

Iterate over a dictionary – use a dictionary itself as an iterator for its keys:

```
>>> for aKey in myDict:  #a feature of later versions of Python
    print 'A %s is %s.' % (aKey, myDict[aKey])
A dog is friendly and smart.
A cat is furry and cute.
```

Iterate over a file:

```
>>> aFile = file('K:\STSCI4060\colors1.txt')
>>> for aLine in aFile:
            print aLine
red
yellow
blue
green
```

# The *for* Statement (cont'd)

Unpack list or tuple elements with a *for* statement:

```
>>>  c1 = [('apple', 'red'), ('banana', 'yellow'), ('kiwi', 'green')]
>>>  for name, color in c1:
            print '%s is %s.' % (name.capitalize(), color)
Apple is red.
Banana is yellow.
Kiwi is green.

>>>c1 = [('Spring', 'Beautiful'), ('Summer', 'Hot'), ('Fall', 'Cool'), ('Winter', 'Cold')]
```

Process items in a sequence with a *for* statement:

```
>>> a = [11, 22, 33]
>>> b = [111, 222, 333]
>>> for idx, value in enumerate(a):  #use a built-in function, enumerate ()
            b[idx] += value
>>> a
[11, 22, 33]
>>> b
[122, 244, 366]
```

# The *for* Statement (cont'd)

Use the **range()** function with the *for* statement:

```
>>> a = [11, 22, 33]
>>> b = [111, 222, 333]
>>> r = range(3)
>>> for i in r:
        b[i]=a[i]+b[i]
>>> a
[11, 22, 33]
>>> b
[122, 244, 366]
```

Nested *for* statements:

```
>>> for x in range(1, 6):
        for y in range(1, 11):
            print '%d * %d = %d.' % (x, y, x*y)
1 * 1 = 1.
1 * 2 = 2.
1 * 3 = 3.
1 * 4 = 4.
1 * 5 = 5.
1 * 6 = 6.
1 * 7 = 7.
1 * 8 = 8.
1 * 9 = 9.
1 * 10 = 10.
2 * 1 = 2.
2 * 2 = 4.
2 * 3 = 6.
2 * 4 = 8.
2 * 5 = 10.
2 * 6 = 12.
2 * 7 = 14.
2 * 8 = 16.
2 * 9 = 18.
2 * 10 = 20.
3 * 1 = 3.
3 * 2 = 6.
3 * 3 = 9.
3 * 4 = 12.
3 * 5 = 15.
3 * 6 = 18.
3 * 7 = 21.
3 * 8 = 24.
3 * 9 = 27.
3 * 10 = 30.
4 * 1 = 4.
4 * 2 = 8.
4 * 3 = 12.
4 * 4 = 16.
4 * 5 = 20.
4 * 6 = 24.
4 * 7 = 28.
4 * 8 = 32.
4 * 9 = 36.
4 * 10 = 40.
5 * 1 = 5.
5 * 2 = 10.
5 * 3 = 15.
5 * 4 = 20.
5 * 5 = 25.
5 * 6 = 30.
5 * 7 = 35.
5 * 8 = 40.
5 * 9 = 45.
5 * 10 = 50.
```

Xiaolong Yang © 2019

# The *while* Statement

The *while* statement is another repeating statement (***while loop***). It executes a block of code until a condition is false.

```
#an example of a while loop

teaTemp = input('Please input the temperature of your tea: ')
while teaTemp > 112:
    print "The temperature of your tea is %d; " % teaTemp, "it's too hot to drink!"
    teaTemp = teaTemp - 1
print "The temperature of your tea is %d. " % teaTemp, "Now your tea is cool enough; you may drink it. "

>>> ================================ RESTART ================================
>>>
Please input the temperature of your tea: 120
The temperature of your tea is 120;  it's too hot to drink!
The temperature of your tea is 119;  it's too hot to drink!
The temperature of your tea is 118;  it's too hot to drink!
The temperature of your tea is 117;  it's too hot to drink!
The temperature of your tea is 116;  it's too hot to drink!
The temperature of your tea is 115;  it's too hot to drink!
The temperature of your tea is 114;  it's too hot to drink!
The temperature of your tea is 113;  it's too hot to drink!
The temperature of your tea is 112.  Now your tea is cool enough; you may drink it.
```

# The *while* Statement (cont'd)

Another example of a *while loop* : calculating class average

The code:

```
''' calculating class average with counter-controlled repetition '''

# initialization
total = 0
scoreCounter = 1
average = 0

# the while loop
while scoreCounter <= 10:
    score = input('Enter a score: ')
    total += score
    scoreCounter +=  1

# calculate the class average
average = total/(scoreCounter - 1)
print 'The class average is' ,average
```

A result:

```
Enter a score: 98
Enter a score : 76
Enter a score : 71
Enter a score : 87
Enter a score : 83
Enter a score : 90
Enter a score : 57
Enter a score : 79
Enter a score : 82
Enter a score : 94
The class average is 81
```

# The *try: … except:* Statements

This pair of statements is to catch exceptions, errors detected during program execution. See the examples below.

```
>>> 20 * (10/0)
Traceback (most recent call last): File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 5 + Cornell*3
Traceback (most recent call last): File "<stdin>", line 1, in ?
NameError: name ' Cornell' is not defined
>>> '5' + 2
Traceback (most recent call last): File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

These are all built-in exceptions, but you can write programs to handle selected exceptions. See some examples of using the *try: … except:* statements for this purpose.

# Example 1: the *try: … except:* Statements

Ask the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-C).

```
>>> while True:
        try:
            x = int(raw_input("Please enter a number: "))
            break
        except ValueError:
            print "Oops! That was no valid number. Try again... "
Please enter a number: a
Oops! That was no valid number. Try again...
Please enter a number: b
Oops! That was no valid number. Try again...
#Use Control-C to interrupt
Please enter a number:
Traceback (most recent call last):
  File "<pyshell#17>", line 3, in <module>
    x = int(raw_input("Please enter a number: "))
KeyboardInterrupt
#exit normally when a number is entered
```

# Example 2: the *try: … except:* Statements

The program:

```
# An exception handling example

number1 = raw_input('Enter numerator:')
number2 = raw_input('Enter denominator:')

try:
    number1 = float(number1)
    number2 = float(number2)
    result = number1/number2

except ValueError:
    print 'You must enter two numbers.'

except ZeroDivisionError:
    print 'Attempt to divide by zero.'

else:
    print '%f / %f = %f' % (number1, number2, result)
```

Some results:

```
Enter numerator:12
Enter denominator:11
12.000000 / 11.000000 = 1.090909

Enter numerator:100
Enter denominator:hello
You must enter two numbers.

Enter numerator:100
Enter denominator:0
Attempt to divide by zero.
```

Note: See **Built-in Exceptions** at https://docs.python.org/2/library/exceptions.html#bltin-exceptions

# Python User-Defined Exceptions

A user-defined exception is derived from the Exception class. The new exception can be raised using the **raise statement** with an optional error message.

```
# define user-defined exceptions
class Error(Exception): #the base class
    pass

class SmallValueError(Error): #subclass raised when value is too small
    pass

class LargeValueError(Error): #subclass raised when value is too large
    pass

number = 5
while True:
   try:
       num = input("Please enter a number: ")
       if num < number:
           raise SmallValueError
       elif num > number:
           raise LargeValueError
       break
   except SmallValueError:
       print "Your number is too small, try again!"
   except LargeValueError:
       print "Your number is too large, try again!"

print "You entered the number correctly!"
```

Xiaolong Yang © 2019

# Python User-Defined Exceptions: The Output

```
Please enter a number: 1
Your number is too small, try again!
Please enter a number: 3
Your number is too small, try again!
Please enter a number: 9
Your number is too large, try again!
Please enter a number: 6
Your number is too large, try again!
Please enter a number: 5
You entered the number correctly!
```