“Selected Programming Language: Introduction to Python”

CS 427

Faculty of Science, Department of Mathematics

Ref:

Richard L. Halterman, LEARNING TO PROGRAM WITH PYTHON, 2011

Richard L. Halterman, Fundamentals of Python Programming, 2017

http://python.cs.southern.edu/pythonbook/pythonbook.pdf

Chapter 8*

More on Functions

Lambda Expressions

- Ordinarily, in order to call a function, we must know its name.
- Invoking functions without using their names directly.

```
def evaluate(f, x, y):
    return f(x, y)
```

- The evaluate function calls f. The name f refers to one of *evaluate*'s parameters; there is no separate function named f specified by *def*
- Python supports the definition of simple, anonymous functions via lambda expressions. The general form of a lambda expression is

```
lambda parameterlist : expression
```

Lambda Expressions

```
>>> def evaluate(f, x, y):
...     return f(x, y)
...
>>> evaluate(lambda x, y: 3*x + y, 10, 2)
32
>>> evaluate(lambda x, y: print(x, y), 10, 2)
10 2
>>> evaluate(lambda x, y: 10 if x == y else 2, 5, 5)
10
>>> evaluate(lambda x, y: 10 if x == y else 2, 5, 3)
2
```

Assignments are not possible within lambda expressions, and loops are not allowed.

A lambda expression can involve one or more function invocations.

Lambda Expressions

- A lambda expression can involve one or more function invocations.
- The lambda expression in the following statement is legal:

```
evaluate(lambda x, y: max(x, y) + x - sqrt(y), 2, 3)
```

- One interesting aspect of lambda functions is that they form what is known as a closure.
- A closure is a function that can capture the context of its surrounding environment.
- Listing 8.15 (closurein.py) demonstrates a simple closure.

Lambda Expressions

Listing 8.15: closurein.py

```
def evaluate(f, x, y):
    return f(x, y)


def main():
    a = int(input('Enter an integer:'))
    print(evaluate(lambda x, y: False if x == a else True, 2, 3))


main()
```

- The main function's local variable a is not passed as a parameter; instead, it is embedded within the lambda code of the first parameter. The variable a is encoded into the lambda expression.

Lambda Expressions

- For an example of a closure transporting a captured local variable out of a function, which includes a function that returns a function (lambda expression) to its caller.

Listing 8.16: makeadder.py

```
def make_adder():
    loc_val = 2   # Local variable definition
    return lambda x: x + loc_val   # Returns a function


def main():
    f = make_adder()
    print(f(10))
    print(f(2))


main()
```

Lambda Expressions

- We can assign a variable to a lambda expression:

```
>>> f = lambda x: 2*x
>>> f(10)
20
```

- It is equivalent to:

```
def f(x):
    return 2*x
```

- We can define an anonymous function and invoke it immediately;

```
print((lambda x, y: x * y)(2, 3))
```

Chapter 9*

Objects

Objects

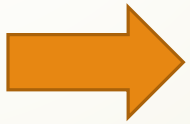
- Python is object oriented. Most modern programming languages support object-oriented (OO) development to one degree or another. An OO programming language allows the programmer to define, create, and manipulate objects.
- Objects bundle together data and functions.
- Like other variables, each Python object has a *type*, or *class*. The terms class and type are synonymous.
- In object-oriented programming, rather than treating data as passive values and functions as active agents that manipulate data, we fuse data and functions together into software units called objects.

String Objects

- The general form of a method call is

Listing 9.2: rjustprog.py

```
word = "ABCD"
print(word.rjust(10, "*"))
print(word.rjust(3, "*"))
print(word.rjust(15, ">"))
print(word.rjust(10))
```



```
******ABCD
ABCD
>>>>>>>>>>>ABCD
          ABCD
```

str Methods
`upper` Returns a copy of the original string with all the characters converted to uppercase
`lower` Returns a copy of the original string with all the characters converted to lower case
`rjust` Returns a string right justified within an area padded with a specified character which defaults to a space
`ljust` Returns a string left justified within an area padded with a specified character which defaults to a space
`center` Returns a copy of the string centered within an area of a given width and optional fill characters; fill characters default to spaces
`strip` Returns a copy of the given string with the leading and trailing whitespace removed; if provided an optional string, the `strip` function strips leading and trailing characters found in the parameter string
`startswith` Determines if the string parameter is a prefix of the invoking string
`endswith` Determines if the string parameter is a suffix of the invoking string
`count` Determines the number times the string parameter is found as a substring within the invoking string; the count includes only non-overlapping occurrences
`find` Returns the lowest index where the string parameter is found as a substring of the invoking string; returns -1 if the parameter is not a substring of the invoking string
`format` Embeds formatted values in a string using {0}, {1}, etc. position parameters

Listing 9.3: stripandcount.py

```
#  Strip leading and trailing whitespace and count substrings
s = "      ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ       "
print("[", s, "]", sep="")
s = s.strip()
print("[", s, "]", sep="")

#  Count occurrences of the substring "BCD"
print(s.count("BCD"))
```

```
[      ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ       ]
[ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ]
3
```

String Objects

- The str class provides a __getitem__ method that returns the character at a given position within the string.
- Since the method's name begins with two underscores (__), the method is meant for internal class use and not for clients.
- The __getitem__ method is special, as clients can access it via a special syntax:

String Objects

```
>>> s = 'ABCEFGHI'
>>> s
'ABCEFGHI'
>>> s.__getitem__(0)
'A'
>>> s.__getitem__(1)
'B'
>>> s.__getitem__(2)
'C'
```

```
>>> s[0]
'A'
>>> s[1]
'B'
>>> s[2]
'C'
```

- The square brackets when used with an object in the manner shown above invoke that object's __getitem__ method.
- In the case of string objects the integer within the square brackets, known as an index, represents the distance from the beginning of the string from which to obtain a character.

String Objects

- The expressions len(s) and s.__len__() are functionally equivalent. Instead of calling the __len__
- method directly, clients should use the global len function.

```
>>> s
'ABCEFGHI'
>>> s = 'ABCEFGHI'
>>> s
'ABCEFGHI'
>>> len(s)
8
>>> s.__len__()
8
```

File Objects

- Fortunately, Python's standard library has a file class that makes it easy for programmers to make objects that can store data to, and retrieve data from, disk.
- The formal name of the class of file objects we will be using is TextIOWrapper, and it is found in the io module.
- Since file processing is such a common activity, the functions and classes defined in the io module are available to any program, and no import statement is required.

File Objects

- The statement

```
f =  open('myfile.txt', 'r')
```

- creates and returns a file object (literally a TextIOWrapper object) named f. The first argument to open is the name of the file, and the second argument is a mode.
- The open function supports the following modes:
  - 'r' opens the file for reading
  - 'w' opens the file for writing; creates a new file
  - 'a' opens the file to append data to it

File Objects

- If the second argument to the open function is missing, it defaults to 'r', so the statement *f = open(fname)* is equivalent to *f = open(fname, 'r')*
- Once you have a file object capable of writing (opened with 'w' or 'a') you can save data to the file associated with that file object using the write method.
- For a file object named f, the statement *f.write('data')* stores the string '*data*' to the file.

File Objects

- The three statements

```
f.write('data')
f.write('compute')
f.write('process')
```

writes the text 'datacomputeprocess' to the file.

File Objects

- If our intention is to retrieve the three separate original strings, we must add delimiters to separate the pieces of data. Newline characters serve as good delimiters:

```
f.write('data\n')
f.write('compute\n')
f.write('process\n')
```

- This places each word on its own line in the text file. The advantage of storing each piece of data on its own line of text is that it makes it easier to read the data from the file with a for statement. :

File Objects

- We also can read the contents of the entire file into a single string using the file object's read method:

```
contents = f.read()
```

- Given the text file from above, the code

```
in = open('compterms.txt', 'r')
s = in.read()
```

assigns to s the string 'data\ncompute\nprocess\n'.

File Objects

- The open method opens a file for reading or writing, and the read, write, and other such methods enable the program to interact with the file.
- When the executing program is finished with its file processing it must call the close method to close the file properly.
- Failure to close a file can have serious consequences when writing to a file, as data meant to be saved could be lost.
- Every call to the open function should have a corresponding call to the file object's close method.

File Objects

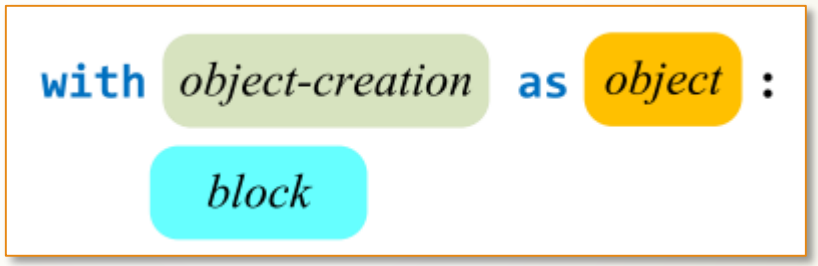
Listing 9.5: simplefileread.py

```
f = open('data.dat')        # f is a file object
for line in f:              # Read each line as text
    print(line.strip())     # Remove trailing newline character
f.close()                   # Close the file
```

- If the file data.dat does not exist or there are issues such as the user does have sufficient permissions to read the file, the executing program will raise an exception.

File Objects

- Since it is important to always close a file after opening it, Python offers a simpler way to automatically closes the file when finished.
- It uses the with/as statement to create what is known as a context manager that ensures the file is closed..



- uses the with/as statement to create what is known as a context manager that ensures the file is closed.

Listing 9.6: simplerread.py

```
with open('data.dat') as f:     # f is a file object
    for line in f:              # Read each line as text
        print(line.strip())     # Remove trailing newline character
    # No need to close the file
```

File Object

TextIOWrapper Methods
`open` A function that returns a file object (instance of `io.TextIOWrapper`).
`read` A method that reads the contents of a text file into a single string.
`write` A method that writes a string to a text file.
`close` A method that closes the file from further processing. When writing to a file, the `close` method ensures that all data sent to the file is saved to the file.

Fraction Objects

- The fractions module provides the Fraction class. Fraction objects model mathematical rational numbers; that is, the ratio of two integers. Rational numbers contain a numerator and denominator.

Listing 9.10: fractionplay.py

```
from fractions import Fraction

f1 = Fraction(3, 4)      # Make the fraction 3/4
print(f1)                # Print it
print(f1.numerator)      # Print numerator
print(f1.denominator)    # Print denominator
print(float(f1))         # Floating-point equivalent
f2 = Fraction(1, 8)      # Make another fraction, 1/8
print(f2)                # Print the second fraction
f3 = f1 + f2             # Add the two fractions
print(f3)                # 3/4 + 1/8 = 6/8 + 1/8 = 7/8
```

Fraction object

```
3/4
3
4
0.75
1/8
7/8
```

Fraction Objects

- The *Fraction(3, 4)* expression returns a reference to the newly created fraction object, and the statement *f1 = Fraction(3, 4)* binds the variable *f1* to this object.
- The expression *f1.numerator* represents the numerator instance variable of object *f1*.
- Python reserves special names for some methods. The Fraction class provides a method named __add__.

```
f3 = f1 + f2
```

```
f3 = f1.__add__(f2)
```

Fraction Objects

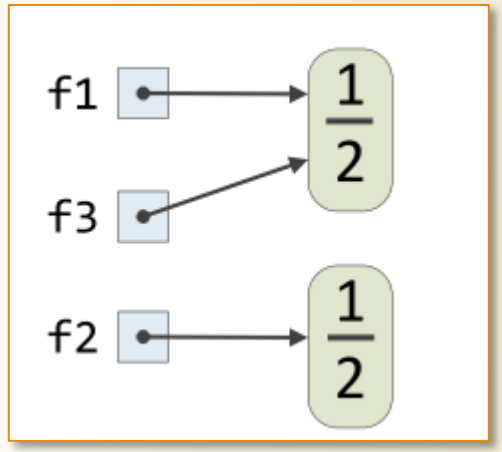
- The Fraction class includes a number of these special methods that exploit syntactic sugar; examples include the following (f and g reference Fraction objects):

- `__mul__`, multiplication: `f.__mul__(g)` is equivalent to `f * g`
- `__eq__`, relational quality: `f.__eq__(g)` is equivalent to `f == g`
- `__gt__`, greater than: `f.__gt__(g)` is equivalent to `f > g`
- `__sub__`, subtraction: `f.__sub__(g)` is equivalent to `f - g`
- `__neg__`, unary minus: `f.__neg__()` is equivalent to `-f`

```
>>> (20).__add__(4)
24
>>> 0.25.__mul__(4)
1.0
>>> 'hello'.__add__('there')
'hellothere'
>>> 'hello'.__mul__(3)
'hellohellohello'
```

Object Mutability and Aliasing

```
#  Assign some Fraction variables
f1 = Fraction(1, 2)
f2 = Fraction(1, 2)
f3 = f1
```



- We have two Fraction objects and three variables bound to Fraction objects. We say that f1 aliases f3.

Aliasing can be an issue for mutable objects.

Thank you