Introduction to Python 3 for People Who Know Many Languages.

By Dan Hyde, July 27, 2011

Python is a free, interpreted language available on multi-platforms including Windows, Mac and Linux. Python has support for multiple programming paradigms including functional, imperative and object-oriented programming.

Follow the "Getting Started with Python 3 on Linux" handout to edit, and run Python 3 programs. Available at
http://www.eg.bucknell.edu/~hyde/Python3/Python3GettingStarted.pdf

Read the following modified six-page introduction to Python by Tim Love at
http://www.eg.bucknell.edu/~hyde/Python3/TimLove.pdf

Notice that Tim originally used Python 2 and Dan has updated the notes for Python 3.

The following notes are written to supplement Tim's web introduction.

Ken Lambert has a web page of the important differences between Python 2 and Python 3 (at introductory CS course level) at the following:
http://home.wlu.edu/~lambertk/python/cs12python/index.html

1. Program Structure

Typically, one imports modules at the top of the program, follows with `def` statements to define functions and then statements.

Python uses indention to show structure. The number of spaces indented is enforced within a structure. When Python detects an outdent, the structure is complete. This enforcing of indents may seem strange at first but it eliminates a lot of Java-style curly braces and makes student programs more readable!

In this case, the first executable line is the assignment to `v_list`.

Since Python is dynamically typed, one doesn't declare the types of variables. One just uses them.

```
# Simple Python program to show program structure
import random    # import a module

def pick_verb(verbs):  # function definition. Note ':' at line's end

    verb = random.choice(verbs) # Python uses indention to show
                                # structure.  Amount of indent
                                # is enforced within a structure.
    return verb

v_list = ['ran', 'jumped', 'walked']
a_verb = pick_verb(v_list)
print(a_verb)

# Since Python is dynamically typed, we can call pick_verb with any sequence.
a = [1, 2, 3, 4, 5]
b = pick_verb(a)
print(b)
```

2. Data

Python's elemental types include numbers (123, 3.1415, 3+4j, and others), strings, Booleans, and others. Composite data types include lists, dictionaries, tuples, and sets. There is support for files and threads. Some simple examples:

Strings: `'Hello!' "can't"  """multiple line string"""`

Lists: `[1,[2, 'three'],4]   [x**2 for x in a if x > 3] # list comprehension`

Dictionaries: `{'food': 'pizza', 'taste': 'yum'}   # Note placement of ':'s and ','s.`

Tuples: `(1, 'spam', 4, 'U')    # Note use of parentheses.`

Sets: `set('abc')  {a, b, c}    # Note use of braces.`

A *sequence* in Python is a positionally ordered collection of other objects. Built-in sequences include strings, lists, dictionaries and tuples and they all follow a consistent set of operations:

```
>>> S = 'Spam'
>>> len(S)     # Length
4
>>> S[0]       # The first item in S, indexing by zero-based position
'S'
>>> S[-1]      # The last item from the end of S
'm'
>>> S[1:3]     # Slice of S from offsets 1 through 2 (not 3)
'pa'
>>> S[1:]      # Everything past the first, i.e., S(1:len(S))
'pam'
>>> S[:3]      # Same as S[0:3]
'Spa'
```

Every object in Python is classified as either immutable (unchangeable) or not. For core types, numbers, strings, tuples, and sets are immutable; lists and dictionaries are not, i.e., they can be changed in-place freely.

Files: Files are fully supported. Here is an example:

```
""" Python program to demo files. By Dan Hyde, May 13, 2011
    It reads a text file a line at a time, strips off trailing
    white space then converts the line of "words" to a list. After
    reversing the list, it converts the list to a string then prints it. """

file_id = open("text1.txt")
line = file_id.readline()

while line:
    line = line.rstrip()       #Strips trailing whitespace
    first_line = list(line)    #Convert "words" to a list
    first_line.reverse()       #Reverse the list
    print("".join(first_line)) #Convert the list to a string
    line = file_id.readline()

file_id.close()
```

3. Sample Program Using User Input, Exception Handling, and Formatted Output

The below `sample.py` file demos reading user input, simple exception handling, and formatted output. The top and interspersed strings are docstrings and are discussed in a later section.

```
"""Python program sample.py to demo user input, exception handling, and
   formatted output.  By Dan Hyde, May 13, 2011"""

def input_int(prompt):
    """ input_int function: prompts the user and converts
        the user's input to a positive integer.
           in: a string for prompt
           out: user input as an integer"""

    while True:
        str_value = input(prompt)  #Input as string
        try:                 #int may raise an exception
            value = int(str_value) #Convert string to integer
            return value
        except Exception:    #Catch any user application exception
            print(str_value + ' is not an integer.  Please retype!')

def convert_quarters(n):
    """convert_quarters function: converts quarters to dollars.
           in: n the number of quarters
           out: value of quarters in dollars"""

    return 25 * n / 100

quarters = input_int('Enter number of quarters: ')
money = convert_quarters(quarters)
print('Amount of money is $%.2f' % (money)) # The second % is string
                                            # formatting op
```

4. Functional Programming Features

In Python, functions are first class objects and can be passed as arguments. But functional programming is more than programming with functions. The key characteristic of a program developed in a functional programming style is that it creates new values by a *transformation*. Usually these values are presented as lists or as dictionaries, i.e., a collection of key:value pairs. Python has flexible implementation of lists and dictionaries to support the functional programming style.

By emphasizing transformation, rather than modification, functional programs work on a larger scale. Transformations are often more uniform and much simpler to write and debug. Errors, when they do occur, tend to be large and thus easier to find and eliminate.

The three most common varieties of transformation are *mapping*, *filtering* and *reduction*. Python has a built-in function for each.

A *mapping* is a one-to-one transformation. Each element in the source is converted into a new value. For example, Python's `map` function contains the transformation in the first argument and a sequence, such as a list, in the second. We need the `list` because in Python 3 `map` is a value generator.

```
>>> print(list(map(abs, [-5,-42, 20, -1])))
[5, 42, 20, 1]
```

A *filtering* is the process of testing each value in a list with a function and retaining only those for which the function is true. The `filter` function requires an argument that is a *predicate*, i.e., a one argument function that returns a Boolean. In Python 3, the function `filter` returns a value generator which we gather together by `list`.

```
def even(x):
    return x % 2 == 0

>>> a = [1, 2, 3, 4, 5]
>>> print(list(filter(even, a)))
[2, 4]
```

A *reduction* is the process of applying a binary function (operation) to each member of a list in a cumulative fashion. Reducing [1, 2, 3, 4] on addition would be the result of (((1+2)+3)+4) or 10.

```
def add2(x, y)
    return x + y

>>> import functools   # need to import special module to use reduce
>>> a = [1, 2, 3, 4]
>>> print(functools.reduce(add2, a))
10
```

If the function is simple, Python has a mechanism called `lambda` to define a nameless function as an expression. Redoing the previous reduction using lambda, we have the following:

```
>>> a = [1, 2, 3, 4]
>>> print(functools.reduce(lambda x, y : x + y, a))
10
```

Python has *list comprehensions* that are easier to read and understand than combinations of `map` and `filter` in part because they do not require an explicit `lambda` function. A list comprehension is a way to create a list by a process that includes a test to filter values.

The list expression `[expr1 for var in list if expr2]` means "Create a list using expression `expr1` for all `var` in source `list` if `expr2` is true." A list comprehension combines aspects of map and filter without the need for a lambda function. The if part is optional.

```
>>> list1 = [1, 2, 3, 4, 5]
>>> print([x*2 for x in list1 if x < 4])
[2, 4, 6]
```

The list comprehension `[x*2 for x in list1 if x < 4]` means to create a list using `x*2` for all `x` in `list1` if `x < 4`.

List comprehensions are often used as the body of a function. The function definition provides a convenient syntax and a way to provide names to arguments. The list comprehension is an easy-to-understand way to write the body of the function:

```
def listOfSquares(a):
    return [x*x for x in a]
>>> listOfSquares([1, 2, 3])
[1, 4, 9]
```

A recursive insertion sort in Python.

```
""" Python program that uses functional programming style
    for an insertion sort.  From page 132 in "Exploring Python"
    by Timothy A. Budd.  Coded by Dan Hyde, May 13, 2011 """

def insertion(aList, x):
    " insert x in proper place in list aList"

    if not aList:   # that is, if aList is empty
        return [x]
    elif x < aList[0]:
        return [x] + aList
    else:
        return aList[0:1] + insertion(aList[1:], x)

def insertionSort(aList):
    """ If aList has values, call recursively.
        An insertion sort of empty list is empty list."""

    if aList:  # if aList has values
        return insertion(insertionSort(aList[1:]), aList[0])
    else:
        return []

p = [11, 3, 6, 2, 7, 12]
q = insertionSort(p)
print(q)
```

A quicksort program that uses list comprehensions.

```
""" A quicksort program that uses list comprehensions.
    From page 134 in "Exploring Python" by Timothy A. Budd.
    Coded by Dan Hyde, May 13, 2011."""

def quicksort(aList):
    """ We chose the middle element for the pivot. """

    if aList:  # i.e., not an empty list
        pivot = aList[len(aList)//2]  # In Python 3, // is integer divide.
        return (quicksort([x for x in aList if x < pivot]) +
                [x for x in aList if x == pivot] +
                quicksort([x for x in aList if x > pivot]))
    else:
        return []

p = [3, 5, 6 , 7 , 1, 12, 9]
q = quicksort(p)
print(q)
```

5. Object-oriented Features

If one has written classes in Java, writing classes in Python is straight forward.

```
""" A BankAccount class in Python.
    By Dan Hyde, May 13, 2011 """

class BankAccount(object):  # Create a new class with object being parent class.
    " A class to model a simple bank account. "

    def __init__(self):    # A constructor uses built-in name __init__
        self.balance = 0.0 # You need to assign a value
                           # to each instance variable.

    def deposit(self, amount):  # In a method "self" is first parameter.
        self.balance = self.balance + amount # Must use "self" on each
                                             # occurrence of instance variable.

    def withdraw(self, amount):
        self.balance = self.balance - amount

    def getBalance(self):
        return self.balance

    def __str__(self):     # Uses built-in name __str__
        """ Method used if object appears in a print function.
            Similar to "toString" method in Java. """

        return 'Balance on account: $' + str(self.balance)

myAccount = BankAccount()    # Make instance of BankAccount
sallyAccount = BankAccount() # Second instance
print("Sally's " + str(sallyAccount)) # The "str" function used to convert
                                      # number to string.

myAccount.deposit(200)   # Called with ONE argument because the receiver for
                         # the message is implicitly passed to the first
                         # argument, i.e., self.
myAccount.withdraw(50)
print("Dan's " + str(myAccount))
```

Output of bank account program:

```
Sally's Balance on account: $0.0
Dan's Balance on account: $150.0
```

Python allows multiple inheritance. Also, one can overload operators like "+" by using special built-in names, in this case, `__add__`.

6. Program Documentation

In addition to comments, Python supports a javadoc-like mechanism for providing documentation on functions, classes, methods and modules. Syntactically, a *docstring* is simply a string that appears immediately after the line containing the `def` in a function or method , or the keyword `class` in a class description, or at the beginning of a file for a module. Note a docstring may be a multi-line string.

```
"""Python program sample.py to demo user input, exception handling, and
   formatted output.  By Dan Hyde, May 13, 2011"""

def input_int(prompt):
    """ input_int function: prompts the user and converts
        the user's input to a positive integer.
           in: a string for prompt
           out: user input as an integer"""

    while True:
        str_value = input(prompt)  #Input as string
        try:                  #int may raise an exception
            value = int(str_value) #Convert string to integer
            return value
        except Exception:    #Catch any user application exception
            print(str_value + ' is not an integer.  Please retype!')

def convert_quarters(n):
    """convert_quarters function: converts quarters to dollars.
           in: n the number of quarters
           out: value of quauters in dollars"""

    return 25 * n / 100

quarters = input_int('Enter number of quarters: ')
money = convert_quarters(quarters)
print('Amount of money is $%.2f' % (money)) # The second % is string
                                            # formatting op
```

Docstrings are great for student documentation. We will want to formulate a CSCI 203/204 standard. To see all the docstrings in a program, we use the built-in `help()` function as follows:

```
>>> import sample
Enter number of quarters: 6
Amount of money is $1.50
>>> help(sample)
Help on module sample:

NAME
    sample

FILE

/nfs/unixspace/linux/accounts/facultystaff/h/hyde/Python/SummerInstitute/sample.py

DESCRIPTION
    Python Program to demo user input, exception handling, and
    formatted output.  By Dan Hyde, May 13, 2011

FUNCTIONS
    convert_quarters(n)
        convert_quarters function: converts quarters to dollars.
```

```
            in: n the number of quarters
            out: value of quarters in dollars

        input_int(prompt)
            input_int function: prompts the user and converts
            the user's input to a positive integer.
               in: a string for prompt
               out: user input as an integer

    DATA
        money = 1.5
        quarters = 6
```

One advantage of docstrings is that they are recognized by the Python interpreter and stored along with the objects as an attribute named `__doc__`. This attribute can be read at run-time:

```
>>> import sample
>>> print(sample.__doc__)
 Python Program to demo user input, exception handling, and
    formatted output.  By Dan Hyde, May 13, 2011
>>> print(sample.input_int.__doc__)
 input_int function: prompts the user and converts
        the user's input to a positive integer.
           in: a string for prompt
           out: user input as an integer
```

The developers of Python try to include readable docstrings in all their modules, classes, method and functions. For example, try the following:

```
>>> import random
>>> print(random.__doc__)
```

When you use the Python `help()` feature, it reads and formats the docstrings as well structural information, e.g., function call patterns, found in a module, class, method or function. Also the tool PyDoc can read the docstrings and structural information to create and format nice looking html web pages.

For document on Python, visit URL: http://www.python.org/doc/