03-statements

January 27, 2017

1 Python Statements and Predicates

2 Assignment

- variables hold arbitrary object references
- objects have a type, variables do not
- in languages like Java/C++, variables are 'created' by declaring them
- in Python, variables are created by assignment
- value(right hand side) is not printed by the read-eval-print loop

```
In [4]: # x gets a reference to the object generated
        # by the expression on the right hand side
        # value of x is not printed

        x = list(range(5))

In [5]: # have to 'eval' x to see what object it refers to

        x

Out[5]: [0, 1, 2, 3, 4]

In [6]: # y gets a reference to the object that x refers to

        y = x

In [7]: # 'x is y ' predicate - are x & y refering to the same object?
        # 'x == y'  predicate - are x & y 'equivalent'?

        [x, y, x is y, x == y]

Out[7]: [[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], True, True]

In [8]: # y gets a new list object

        y = list(range(5))
        y

Out[8]: [0, 1, 2, 3, 4]
```

```
In [9]: # now x and y point to different objects that are equivalent

        [x,y, x is y, x == y]

Out[9]: [[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], False, True]
```

2.1 Unpacking Assignments

```
In [10]: # can do several assignments in one statement

         x, y, z = 1, 2, 3
         [x, y, z]

Out[10]: [1, 2, 3]

In [11]: # same as above

         (x,y,z) = (4,5,6)
         [x,y,z]

Out[11]: [4, 5, 6]

In [12]: # works with lists as well

         [x, [y, z]] = [7,[8,9]]

         [x, y, z]

Out[12]: [7, 8, 9]

In [13]: # unpacking happens 'in parallel'
         # don't need tmps to do 'swaps'

         y, x = x, y
         [x, y]

Out[13]: [8, 7]

In [14]: # if left and right side don't match, will get an error

         x,y = 1,2,3


        ---------------------------------------------------------------------------
        ValueError                                Traceback (most recent call last)

        <ipython-input-14-45569d1bc41f> in <module>()
          1 # if left and right side don't match, will get an error
          2
```

```
----> 3 x,y = 1,2,3


ValueError: too many values to unpack (expected 2)
```

```
In [15]: # *var will match an arbitrary number of elements, including zero

         head, *tail = [1,2,3,4]
         [head, tail]

Out[15]: [1, [2, 3, 4]]

In [16]: head, *tail = [1,2]
         [head, tail]

Out[16]: [1, [2]]

In [17]: x, *y, z = [1,2,3,4,5]
         [x,y,z]

Out[17]: [1, [2, 3, 4], 5]
```

3 Statement Blocks

- some statements, like 'if', 'for', 'while' and 'def', 'class', 'try' end with a ':' to mark a new block
- subsequent statements in the block must be indented
- the block ends when the indenting reverts to the previous level
- in other words, python demarcates "statement blocks" by indentation. Java/C++ uses '{' '}'
- indentation must be correct, or program will either be incorrect, or not run at all

4 if

- unlike C++/Java, Python doesn't require parens around the predicate
- elif, else claues are optional
- elif is used to "chain" if's (no switch statement in Python)
- else clause is executed if all predicates fail
- Python doesn't have a 'switch' statement - use if like example below

```
In [18]: # note ":" at end of if, elif, else
         # note identing of print statements

         flag = 1

         if flag == 1:
             # this clause will be executed
             print('flag == 1')
```

```
        elif flag == 2:
            print('flag == 2')
        elif flag == 3:
            print('flag == 3')
        else:
            print("flag didn't == 1 or 2 or 3")
        print('end of indent ends if statement')
```

```
flag == 1
end of indent ends if statement
```

```
In [19]: flag = 2

        if flag == 1:
            print('flag == 1')
        elif flag == 2:
            # this clause will be executed
            print('flag == 2')
        elif flag == 3:
            print('flag == 3')
        else:
            print("flag didn't == 1 or 2 or 3")
        print('end of indent ends if statement')
```

```
flag == 2
end of indent ends if statement
```

```
In [20]: flag = 134

        if flag == 1:
            print('flag == 1')
        elif flag == 2:
            print('flag == 2')
        elif flag == 3:
            print('flag == 3')
        else:
            # this 'default clause' will be executed
            print("flag didn't == 1 or 2 or 3")
        print('end of indent ends if statement')
```

```
flag didn't == 1 or 2 or 3
end of indent ends if statement
```

5 Example: decrypt

```
In [21]: # this version uses 'if' statement
```

```
        def decrypt(s):
            words = []
            for j in range(len(s)):
                if s[j].isdigit():
                    wlen = int(s[j])
                    words.append(s[j+1:j+1+wlen])
            return words
```

```
In [22]: e = '{SVIu6Python-)dKct@\\JK)2is:y:=;;~6reallyMZ-&Bk`*6great!NB!|Krj##'
         decrypt(e)
```

```
Out[22]: ['Python', 'is', 'really', 'great!']
```

6 Ternary if

- very useful
- unlike normal if, it is an expression, not a statement, so it returns a value
- like 'pred ? TrueVal : FalseVal' in Java/C/C++
- peculiar syntax

```
In [23]: predicate = True

         val = 'true val' if predicate else 'false val'
         val
```

```
Out[23]: 'true val'
```

```
In [24]: predicate = False

         val = 'true val' if predicate  else 'false val'
         val
```

```
Out[24]: 'false val'
```

7 for

- basic way to iterate, but not always the best
- iterates over the elements of a list
- later we will learn about the "iteration protocol"
- note ":" and indentation

```
In [25]: for x in [3,6,7,2]:
             # body of the for
             print(x)
             print(x+10)
```

```
3
13
```

```
6
16
7
17
2
12
```

8 'for' helper functions

- 'range'
- 'enumerate'
- 'zip'

```
In [26]: # range - 'for' will iterate over list specified by range

         sum = 0

         for n in range(2, 7, 2):
             print('element', n)
             sum += n

         print('sum',sum)
```

```
element 2
element 4
element 6
sum 12
```

```
In [27]: # if you are iterating over an arbitrary list,
         # as opposed to a range, there is no index
         # 'enumerate' adds an index

         x = ['mudd', 'shapiro', 'butler']
         enumerate(x)
```

```
Out[27]: <enumerate at 0x106cdf990>
```

```
In [28]: # enumerate is lazy!
         # use list to force evaluation
         # get a length 3 list where each element is a length 2 tuple

         list(enumerate(x))
```

```
Out[28]: [(0, 'mudd'), (1, 'shapiro'), (2, 'butler')]
```

```
In [29]: # note 'j, b' - destructures the length 2 tuples
```

```
        for j, b in enumerate(x):
            print(j, b)
```

```
0 mudd
1 shapiro
2 butler
```

In [11]:
```
### decrypt version from above

def decrypt(s):
    words = []
    for j in range(len(s)):
        if s[j].isdigit():
            wlen = int(s[j])
            words.append(s[j+1:j+1+wlen])
    return words

# a better version uses enumerate

def decrypt2(s):
    words = []
    for j, c in enumerate(s):
        if c.isdigit():
            wlen = int(s[j])
            words.append(s[j+1:j+1+wlen])
    return words

# can do even better with list comprehension

def decrypt3(s):
    # does not use if statement
    return [s[j+1:j+1+int(s[j])] for j, c in enumerate(s) if c.isdigit()]
```

In [12]:
```
e = '{SVIu6Python-)dKct@\\JK)2is:y:=;;~6reallyMZ-&Bk`*6great!NB!|Krj##'
[decrypt2(e), decrypt3(e)]
```

Out[12]:
```
[['Python', 'is', 'really', 'great!'], ['Python', 'is', 'really', 'great!'
```

In [30]:
```
x
```

Out[30]:
```
['mudd', 'shapiro', 'butler']
```

In [31]:
```
# sometimes you want to iterate thru two or more lists simultaneously
# 'zip' - threads lists together. 'zip' is lazy
# another list of tuples

r = range(3)
y = ['engineering', 'compsci', 'library']
list(zip(r, x, y))
```

```
Out[31]: [(0, 'mudd', 'engineering'),
          (1, 'shapiro', 'compsci'),
          (2, 'butler', 'library')]

In [32]: # index, name, func destructures the tuples

         for index, name, func in zip(r, x, y):
             print(index, name, func)

0 mudd engineering
1 shapiro compsci
2 butler library


In [33]: # mix it up

         list(enumerate(zip(x, y)))

Out[33]: [(0, ('mudd', 'engineering')),
          (1, ('shapiro', 'compsci')),
          (2, ('butler', 'library'))]

In [34]: # 'p' is bound to the 2 element tuple from the zip

         for j, p in enumerate(zip(x, y)):
             print(j, p)

0 ('mudd', 'engineering')
1 ('shapiro', 'compsci')
2 ('butler', 'library')


In [35]: # directly match the structure

         for j,[a,b] in enumerate(zip(x, y)):
             print(j, a, b )

0 mudd engineering
1 shapiro compsci
2 butler library
```

9 Set Comprehensions

```
In [36]: # accumulate to a set
         # 'add' to a set, not 'append'
         # duplicates eliminated

         result = set()
```

```
for x in [3,11,2,3,11,14]:
    if x > 10:
        result.add(x*10)
result
```

```
Out[36]: {110, 140}
```

```
In [37]: # better - use a 'set comprehension'

s = {x*10 for x in [3,11,2,3,11,14] if x>10}
[s, type(s)]
```

```
Out[37]: [{110, 140}, set]
```

10 Dict comprehensions

```
In [38]: d = {}

for x in range(5):
    d[x] = x+10

d
```

```
Out[38]: {0: 10, 1: 11, 2: 12, 3: 13, 4: 14}
```

```
In [39]: # dict comprehension

{x:x+10 for x in range(5)}
```

```
Out[39]: {0: 10, 1: 11, 2: 12, 3: 13, 4: 14}
```

11 while

- used for more complex loops that depend on arbitrary conditions to terminate
- 'break' and 'continue' work in for/while loops
- Python does not have var++, ++var, var–, –var

```
In [40]: n = 0
while n < 7:
    print(n)
    n += 1
```

```
0
1
2
3
4
5
6
```

```
In [41]: n = 0
         while n < 7:
             n += 1
             if n == 2:
                 continue
             print(n)
             if n > 4:
                 break
```

```
1
3
4
5
```

```
In [42]: # proposed in 1937
         # conjecture is the sequence always
         # terminates in 1, but nobody has
         # been able to prove it

         def collatz(n):
             seq = [n]
             # keep looping until we get 1
             while n != 1:
                 if n % 2 == 0:
                     n = n//2
                 else:
                     n = 3*n + 1
                 seq.append(n)
             return seq
```

```
In [43]: collatz(6)
```

```
Out[43]: [6, 3, 10, 5, 16, 8, 4, 2, 1]
```

```
In [44]: collatz(19)
```

```
Out[44]: [19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4,
```

```
In [45]: print(collatz(27))
```

```
[27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364,
```

12 del

- used to 'delete' various things

```
In [46]: # will remove a variable binding...

         x = 'foo'
         y = x
         x

Out[46]: 'foo'

In [47]: del x

         x


         ---------------------------------------------------------------------------

         NameError                                 Traceback (most recent call last)

         <ipython-input-47-83adaf4e6389> in <module>()
           1 del x
           2
     ----> 3 x


         NameError: name 'x' is not defined


In [48]: # but 'del' does NOT remove the 'foo' string object
         # objects ONLY disappear when there are NO references
         # to them left

         y

Out[48]: 'foo'

In [49]: # make a small dict

         d = dict()
         d[3] = 33
         d[4] = 44
         d

Out[49]: {3: 33, 4: 44}

In [50]: # delete a key/value pair

         del d[3]
         d

Out[50]: {4: 44}
```

```
In [51]: x = list(range(10))
         x

Out[51]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [52]: # delete a slice from a list

         del x[3:7]
         x

Out[52]: [0, 1, 2, 7, 8, 9]
```

13 pass

- just a statement placeholder - does absolutely nothing

```
In [53]: if True:
             pass

         print('got here')

got here
```

14 import

- a module is a set of one or more files of python code
- 'importing' a module loads that code into Python and makes that functionality available
- similar to the Java package system
- several types of imports
- executable statement - not a declaration

```
In [54]: # 'choice' is a function in the 'random' module
         # but choice is not available,
         # because the module isn't loaded

         random.choice


        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-54-58938c6af127> in <module>()
          3 # because the module isn't loaded
          4
    ----> 5 random.choice
```

```
        NameError: name 'random' is not defined


In [55]: # this makes names in random available, but the names
         # must prefixed with 'random.'

         import random
         [random.choice, random.shuffle, random.sample]

Out[55]: [<bound method Random.choice of <random.Random object at 0x103821018>>,
          <bound method Random.shuffle of <random.Random object at 0x103821018>>,
          <bound method Random.sample of <random.Random object at 0x103821018>>]

In [56]: # choice still not defined without qualification

         choice


        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-56-9f39f429a368> in <module>()
          1 # choice still not defined without qualification
          2
    ----> 3 choice


        NameError: name 'choice' is not defined


In [57]: # this makes in random available
         # using a shorter'nickname'

         import random as ran

         [ran.choice, ran.sample, ran.shuffle]

Out[57]: [<bound method Random.choice of <random.Random object at 0x103821018>>,
          <bound method Random.sample of <random.Random object at 0x103821018>>,
          <bound method Random.shuffle of <random.Random object at 0x103821018>>]

In [58]: # choice still is not defined at top level

         choice


        ---------------------------------------------------------------------------
```

```
NameError                                 Traceback (most recent call last)

<ipython-input-58-439384a288fc> in <module>()
  1 # choice still is not defined at top level
  2
----> 3 choice


NameError: name 'choice' is not defined
```

15 from

- imports names to top level

```
In [59]: # now don't need to say 'random.choice', just 'choice'

         from random import choice
         choice

Out[59]: <bound method Random.choice of <random.Random object at 0x103821018>>

In [60]: # another function name in random, still not defined
         # at top level

         shuffle


         ---------------------------------------------------------------------------

         NameError                                 Traceback (most recent call last)

         <ipython-input-60-3fbeab82b732> in <module>()
           2 # at top level
           3
         ----> 4 shuffle


         NameError: name 'shuffle' is not defined


In [61]: # this puts all the names in random at top level

         from random import *
         [shuffle, choice, sample]

Out[61]: [<bound method Random.shuffle of <random.Random object at 0x103821018>>,
          <bound method Random.choice of <random.Random object at 0x103821018>>,
          <bound method Random.sample of <random.Random object at 0x103821018>>]
```

16 Generalized booleans

- it is convenient to generalize what is considered to be True and False
- None, 0, and empty collections(strings, lists, tuples, dictionaries, sets), are equivalent to False
- Any other object is equivalent to True

```
In [62]: # list of things to try

         x = [0, 1, "", "stuff", {}, {3:5}, {3,5}, (), (1,2), None]

         for e in x:
             # ternary if is an expression,
             # so can be an arg to print
             print(e, True if e else False)
```

```
0 False
1 True
 False
stuff True
{} False
{3: 5} True
{3, 5} True
() False
(1, 2) True
None False
```

17 short circuit evaluation of booleans

- 'and' and 'or' do 'short circuit' evaluation
- evaluation stops as soon as True/False value is known
- note result is NOT always True or False

```
In [13]: # 'or' eval stops at first True value and returns it

         False or 0 or [] or 6 or 7
```

```
Out[13]: 6
```

```
In [14]: # 'and' eval stops at first False value and returns it

         True and 5 and [3,4] and {} and 34 and 200
```

```
Out[14]: {}
```

18 Example: Filtering and modifying a list

- dir returns a list of methods for a type

- want to get rid of methods with a '__' in the name
- want to capitalize remaining names

```
In [63]: # dir lists methods of a type. want to get rid of methods with a '__'(they

         dir(list)

Out[63]: ['__add__',
          '__class__',
          '__contains__',
          '__delattr__',
          '__delitem__',
          '__dir__',
          '__doc__',
          '__eq__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__getitem__',
          '__gt__',
          '__hash__',
          '__iadd__',
          '__imul__',
          '__init__',
          '__iter__',
          '__le__',
          '__len__',
          '__lt__',
          '__mul__',
          '__ne__',
          '__new__',
          '__reduce__',
          '__reduce_ex__',
          '__repr__',
          '__reversed__',
          '__rmul__',
          '__setattr__',
          '__setitem__',
          '__sizeof__',
          '__str__',
          '__subclasshook__',
          'append',
          'clear',
          'copy',
          'count',
          'extend',
          'index',
          'insert',
```

```
 'pop',
 'remove',
 'reverse',
 'sort']
```

In [64]:
```
# can filter and capitalize with single list comprehension

[s.capitalize() for s in dir(list) if '_' not in s]
```

Out[64]:
```
['Append',
 'Clear',
 'Copy',
 'Count',
 'Extend',
 'Index',
 'Insert',
 'Pop',
 'Remove',
 'Reverse',
 'Sort']
```