Lecture 16 - Python Features

Week 7 Wednesday

Miles Chen, PhD

Taken from Chapter 19 of Think Python by Allen B Downey

Python has a number of features that are not necessary, but with them you can sometimes write code that's more concise, readable, or efficient.

Conditional Expressions

A conditional expression will check a condition and run the associated code.

The following example shows how we can ask Python to find the natural log of a number. logs do not exist for non-positive values, so if x is less than or equal to zero, we want to return `nan` instead of an error.

In [1]:
```
x = -3
```

In [2]:
```
import math

if x > 0:
    y = math.log(x)
else:
    y = float('nan')
y
```

Out[2]:
```
nan
```

We can express the same idea more concisely with a conditional expression.

In [5]:
```
x = math.e
```

In [6]:
```
y = math.log(x) if x > 0 else float('nan')
```

In [7]:
```
y
```

Out[7]:
```
1.0
```

Recursive functions can be rewritten as conditional expressions.

In [8]:
```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

In [9]:
```
factorial(5)
```

Out[9]:
```
120
```

In [10]:
```
def factorial(n):
    return 1 if n == 0 else n * factorial(n - 1)
```

In [11]:
```
factorial(6)
```

Out[11]:
```
720
```

The conditional expression is certainly more concise. Whether it is more readable is debatable.

In general, if both branches of a conditional statement are simple expressions that are assignmened or a returned, it can be written as a conditional expression.

Variable Length Arguments and Key-Word Arguments

When we covered tuples, we saw that you can gather arguments together with *

In [12]:
```
def print_all(*args):
    for a in args:
        print(a)
```

In [13]:
```
print_all(1,2,3,4,5)
```

```
1
2
3
4
5
```

In [14]:
```
from random import randint

def roll(*dice):
    total = 0
    for die in dice:
        roll = randint(1, die)
        print(roll)
        total += roll
    return total
```

In [15]:
```
roll(20)
```

```
18
```

Out[15]:
```
18
```

In [16]:
```
roll(6, 6, 20)
```

```
5
5
5
```

Out[16]: 15

In [21]:
```
roll(6, 6, 20)
```

```
1
3
3
```

Out[21]: 7

In [27]:
```
roll(6, 6, 20, 20)
```

```
5
3
13
3
```

Out[27]: 24

Similarly, you can gather key-word pairs as arguments and create a function that uses them.

In [28]:
```
def print_contents(**kwargs):
    for key, value in kwargs.items():
        print ("key %s has value %s" % (key, value))
```

In [29]:
```
print_contents(CA = "California", OH = "Ohio")
```

```
key CA has value California
key OH has value Ohio
```

In [30]:
```
keys = ['CA', 'OH', 'TX', 'WA']
names = ["California", "Ohio", "Texas", "Washington"]
d = dict(zip(keys, names))
print(d)
```

```
{'CA': 'California', 'OH': 'Ohio', 'TX': 'Texas', 'WA': 'Washington'}
```

In [31]:
```
# if you want to pass a dictionary to the function, you have to use `**` to scatter them
print_contents(d)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-07f6714a297c> in <module>
      1 # if you want to pass a dictionary to the function, you have to use `*
*` to scatter them
----> 2 print_contents(d)

TypeError: print_contents() takes 0 positional arguments but 1 was given
```

In [32]:
```
# if you want to pass a dictionary to the function, you have to use `**` to scatter them
print_contents(**d)
```

```
key CA has value California
key OH has value Ohio
key TX has value Texas
key WA has value Washington
```

In [ ]:
```
# popular use case: matplotlib
# {color = "blue", line_type = 2, line_width = 3}
# you want to make 5 plots all with the same settings
# rather than copy paste the settings into all of the plots,
# make a dictionary with the settings, and pass the dictionary using **kwargs
```

List comprehensions

List comprehensions allow us to create new lists concisely based on an existing collection

They take the form:

```
[expr for val in collection if condition]
```

This is basically equivalent to the following loop:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

In [33]:
```
# make a list of the squares
[x**2 for x in range(1,11)]
```

Out[33]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In [34]:
```
import numpy as np
np.array([x**2 for x in range(1,11)])
```

Out[34]:
```
array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100])
```

In [35]:
```
# square only the odd numbers
[x**2 for x in range(1,11) if x % 2 == 1]
```

Out[35]:
```
[1, 9, 25, 49, 81]
```

In [36]:
```
# take a list of strings, and write the words that are over 2 characters long in up
percase.
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
[x.upper() for x in strings if len(x) > 2]
```

Out[36]:
```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

You can create a list comprehension from any iterable (list, tuple, string, etc)

In [37]:
```
# extract the digits from a string
string = "Hello 963257 World"
[int(x) for x in string if x.isdigit()]
# for x in string, will look at each character individually
# if x is a digit, then convert it using int()
```

Out[37]:
```
[9, 6, 3, 2, 5, 7]
```

In [38]:
```
# iterate over a dictionary's items
d = {'a':'apple', 'b':'banana', 'c':'cookie'}
```

In [39]:
```
list(d.items())  # recall what dict.items() returns: a list of tuples
```

Out[39]:
```
[('a', 'apple'), ('b', 'banana'), ('c', 'cookie')]
```

In [40]:
```
[key + ' is for ' + value for key, value in d.items() if key != 'b' ]
```

Out[40]:
```
['a is for apple', 'c is for cookie']
```

Dictionary Comprehensions

A dict comprehension looks like this:

```
dict_comp = {key-expr : value-expr for value in collection if condition}
```

Look at the list `strings` from above.

In [41]:
```
# create a dictionary, where the key is the word capitalized, and the value is the length of the word
fruits = ['apple', 'mango', 'banana','cherry']
{f.capitalize():len(f) for f in fruits}
```

Out[41]:
```
{'Apple': 5, 'Mango': 5, 'Banana': 6, 'Cherry': 6}
```

In [42]:
```
# create a dictionary where the key is the index, and the value is the string in th
e strings list.
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
```

In [43]:
```
list(enumerate(strings))  # enumerate produces a collection of tuples, with index a
nd value
```

Out[43]:
```
[(0, 'a'), (1, 'as'), (2, 'bat'), (3, 'car'), (4, 'dove'), (5, 'python')]
```

In [44]:
```
index_map = {index:val for index, val in enumerate(strings)}
index_map
```

Out[44]:
```
{0: 'a', 1: 'as', 2: 'bat', 3: 'car', 4: 'dove', 5: 'python'}
```

In [45]:
```
# note that enumerate returns tuples in the order (index, val)
# in the creation of a dictionary, you can swap those positions
# and even apply functions to them

# We create a dictionary where the key is the string, and the value is the index in
the strings list.
loc_mapping = {val : index for index, val in enumerate(strings)}
loc_mapping
```

Out[45]:
```
{'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

In [46]:
```
index_map['a']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-46-a566f0150b5c> in <module>
----> 1 index_map['a']

KeyError: 'a'
```

In [47]:
```
loc_mapping['a']
```

Out[47]:
```
0
```

In [48]:
```
# combine dictionaries with kwargs
dd = {**loc_mapping, **index_map}
print(dd)
```

```
{'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5, 0: 'a', 1: 'as',
2: 'bat', 3: 'car', 4: 'dove', 5: 'python'}
```

In [49]:
```
# even better... use dict.update(). This modifies the dictionary in place
loc_mapping.update(index_map)
loc_mapping
```

Out[49]:
```
{'a': 0,
 'as': 1,
 'bat': 2,
 'car': 3,
 'dove': 4,
 'python': 5,
 0: 'a',
 1: 'as',
 2: 'bat',
 3: 'car',
 4: 'dove',
 5: 'python'}
```

Generator Expressions

Generator Expressions are similar to List comprehensions.

You create them with parentheses instead of square brackets.

The result is a generator object. You can access values in the generator using `next()`

In [50]:
```
g = (n**2 for n in range(10))
```

In [51]:
```
g
```

Out[51]:
```
<generator object <genexpr> at 0x00000243E6A0DF48>
```

In [52]:
```
next(g)
```

Out[52]:
```
0
```

In [53]:
```
next(g)
```

Out[53]:
```
1
```

In [54]:
```
next(g)
```

Out[54]:
```
4
```

In [55]:
```
for val in g:
    print(val)
```

```
9
16
25
36
49
64
81
```

In [56]:
```
next(g) # calling next after it has run out of iterations will result in an error
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-56-e734f8aca5ac> in <module>
----> 1 next(g)

StopIteration:
```

List Comprehension vs Generator Expressions in Python

A Key difference between a list comprehension and a generator is that the generator is lazy.

The list comprehension will evaluate the entire sequence of iterations. The generator will only generate the next value when it is asked to do so.

Depending on the expression that needs to be evaluated, you may prefer to use a generator over the list comprehension.

The following examples are from: [https://code-maven.com/list-comprehension-vs-generator-expression (https://code-maven.com/list-comprehension-vs-generator-expression)](https://code-maven.com/list-comprehension-vs-generator-expression)

In [57]:
```
l = [n*2 for n in range(1000)] # List comprehension
g = (n*2 for n in range(1000))  # Generator expression
```

In [58]:
```
print(type(l))  # 'list'
print(type(g))  # 'generator'
```

```
<class 'list'>
<class 'generator'>
```

In [59]:
```
import sys
print(sys.getsizeof(l))  # more space in memory
print(sys.getsizeof(g))  # less space in memory
```

```
9024
120
```

In [60]:
```
# cannot access values in a generator by index
print(l[4])    # 8
print(g[4])    # TypeError: 'generator' object is not subscriptable
```

```
8

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-60-e29ce47c972b> in <module>
      1 # cannot access values in a generator by index
      2 print(l[4])    # 8
----> 3 print(g[4])    # TypeError: 'generator' object is not subscriptable

TypeError: 'generator' object is not subscriptable
```

In [61]:
```
next(g)
```

Out[61]: 0

In [62]:
```
next(g)
```

Out[62]: 2

In [63]:
```
next(g)
```

Out[63]: 4

In [64]:
```
next(g)
```

Out[64]: 6

In [65]:
```
sum(g)
```

Out[65]: 998988

In [66]:
```
sum(l)
```

Out[66]: 999000

In [ ]:
```

```