Loop Control Statements

There are two major kinds of programming loops:

- Counting loops: In Python, counting loops are often defined with the for statement.
  - The for statement

```
for x in range(10):
    for_body
```

  - The for-else statement

```
for x in range(10):
    for_body
else:
    #reach this block if this for-loop is at the last iteration
```

- Event-controlled loops: In Python, event-controlled loops are defined with the while statement.
  - The while statement

```
while condition:
    while_body
```

  - The while-else statement

```
while condition:
    while_body
else:
    #reach this block if this while-loop is not terminated by the break
statement
```

In [1]:

```
for i in range(3):
    print(i)
else:
    print('finished')
```

```
0
1
2
finished
```

In [2]:

```
for i in range(3):
    if i == 2:
        break
    else:
        print(i)
else:
    print('finished')
```

```
0
1
```

In [3]:

```
total = 0
i = 0
while i <= 3:
    total += i
    print(i,total)
    i += 1
```

```
0 0
1 1
2 3
3 6
```

In [4]:

```
total = 0
i = 0
while i <= 3:
    total += i
    print(i,total)
    i += 1
else:
    print('final:',i,total)
```

```
0 0
1 1
2 3
3 6
final: 4 6
```

The For Statement

We can use for loops to iterate over sequences including ranges, lists, tuples, dictionaries, etc.

In [5]:

```
my_list=[9,10,20,15,5,14]
for i in my_list:
    print(i)
```

```
9
10
20
15
5
14
```

In [6]:

```
my_set = {9,10,20,15,5,14}
for i in my_set:
    print(i)
```

```
5
9
10
14
15
20
```

In [7]:

```
my_tuple=(9,10,20,15,5,14)
for i in my_tuple:
    print(i)
```

```
9
10
20
15
5
14
```

In [8]:

```
my_dict = {9:'a',10:'b',20:'c',15:'d',5:'e',14:'f'}
# iterate over keys
for i in my_dict:
    print(i)
```

```
20
5
9
10
14
15
```

In [9]:

```
# iterate over values
for i in my_dict.values():
    print(i)
```

```
c
e
a
b
f
d
```

In [10]:

```
# iterate over items
for k,v in my_dict.items():
    print(k,v)
```

```
20 c
5 e
9 a
10 b
14 f
15 d
```

In for-loops, if the index and value of the element in a list are required to know, you can use the enumerate function to number the elements:

In [11]:

```
pets = ["cat", "dog", "budgie"]
# only use each element
for pet in pets:
    print(pet)

# use the index to access each element
for i in range(len(pets)):
    print(i, pets[i].upper())

# better Python code
for i, pet in enumerate(pets):
    print(i, pet.upper())
```

```
cat
dog
budgie
0 CAT
1 DOG
2 BUDGIE
0 CAT
1 DOG
2 BUDGIE
```

Be careful when you iterate over a list and remove some elements in this list.

In [12]:

```
#Incorrect version
numbers = [1,2,2,3]
for i, num in enumerate(numbers):
    if num == 2:
        del numbers[i]
print(numbers)
```

```
[1, 2, 3]
```

In [13]:

```
#Correct version
numbers = [1,2,2,3]
for i in range(len(numbers)-1,-1,-1):
    if numbers[i]==2:
        del numbers[i]
print(numbers)
```

```
[1, 3]
```

Iterables, Iterators, and Generators

- An **iterable** is a type which can be iterated over with a for loop. Lists, ranges, tuples, strings, dictionaries, and sets are all iterable types.
  - What is the main difference between ranges and the other five iterable types: lists, strings, dictionaries, tuples, and sets?
    - Ranges are not container types.
    - Ranges do not require memory space to store the element in ranges. Instead, the elements in ranges are yielded by some methods. (Note that range in Python 3.x is xrange in Python 2.x)
- Every iterable object has a method which returns an iterator over that object.
  - An **iterator** is used to access the element in a sequence one by one sequentially.
- In Python, an iterable type that generates values on demand is referred to as a generator.
  - range and enumerate are **generators**.

In [14]:

```
print(range(5)) # range(0,5) because range is a generator
print(list(range(5))) # generate a list [0,1,2,3,4] from range(5)
```

```
range(0, 5)
[0, 1, 2, 3, 4]
```

In [15]:

```
r = (1,2,3,4,5)
for i in r:
    print(i)

for i in range(1,6):
    print(i)
```

```
1
2
3
4
5
1
2
3
4
5
```

In [16]:

```
r = (1,2,3,4,5)
# an iterable object has an instance method __iter__,
# which returns an iterator for accessing each element in this object and can be called by
print(dir(r))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc_
_', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__
getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__le
n__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex_
_', '__repr__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subcl
asshook__', 'count', 'index']
```

In [17]:

```
# an iterator has an instance method __next__, which retrieves the next element in this ite
print(dir(iter(r)))
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__iter__',
'__le__', '__length_hint__', '__lt__', '__ne__', '__new__', '__next__', '__r
educe__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__siz
eof__', '__str__', '__subclasshook__']
```

In [18]:

```
r = (1,2,3,4,5)
p = iter(r)
try:
    while True:
        v = next(p)
        print(v)
except StopIteration: # the iterator is exhausted
    pass;
```

```
1
2
3
4
5
```

In [19]:

```
# i and j are two independent iterators
r = (1,2)
for i in r:
    for j in r:
        print(i,j)
```

```
1 1
1 2
2 1
2 2
```

Some built-in generators defined in the itertools module

```
# import this module in order to use it
import itertools

# unlike range, count doesn't have an upper bound,
# and is not restricted to integers

for i in itertools.count(1):
    print(i) # 1, 2, 3....
for i in itertools.count(1, 0.5):
    print(i) # 1.0, 1.5, 2.0....

# cycle repeats the values in another iterable over and over
for animal in itertools.cycle(['cat', 'dog']):
    print(animal) # 'cat', 'dog', 'cat', 'dog'...

# repeat repeats a single item
for i in itertools.repeat(1): # ...forever
    print(i) # 1, 1, 1....
```

In [20]:

```
# import this module in order to use it
import itertools

for i in itertools.repeat(1, 3): # or a set number of times
    print(i) # 1, 1, 1

# chain combines multiple iterables sequentially
numbers = [1,2,3]
animals = ['cat', 'dog']
for i in itertools.chain(numbers, animals):
    print(i) # print all the numbers and then all the animals

# produce all permutations
for p in itertools.permutations([1,2,3]):
    print(p) # [1,2,3], [1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]
```

```
1
1
1
1
2
3
cat
dog
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

The zip build-in function is used to combine multiple iterables pairwise.

In [21]:

```
a=[1, 2, 3, 4, 5]
b=('a','b','c','d','e')
c=[9,10,11,12,13]
for u,v,w in zip(a,b,c):
    print(u,v,w)
```

```
1 a 9
2 b 10
3 c 11
4 d 12
5 e 13
```

Comprehensions

A comprehension is a kind of filter which we can define on an iterable based on some conditions.

Example 1:

```
numbers = [1, 5, 2, 12, 14, 7, 18]
doubles = []
for number in numbers:
    doubles.append(2*number)
```

List comprehension

```
doubles = [2*number for number in numbers]
```

Example 2:

```
even_numbers = []
for number in numbers:
    if number % 2 == 0:
        even_numbers.append(number)
```

List comprehension

```
even_numbers = [number for number in numbers if number % 2 == 0]
```

Example 3:

```
animals = ['aardvark', 'cat', 'dog', 'opossum']
vowel_animals = []
for animal in animals:
    if animal[0] in 'aeiou':
        vowel_animals.append(animal.title())
```

List comprehension

```
vowel_animals =[animal.title() for animal in animals if animal[0] in 'aeiou']
```

In [22]:

```
numbers = [1, 5, 2, 12, 14, 7, 18]
animals = ['aardvark', 'cat', 'dog', 'opossum']

doubles = [2*number for number in numbers]
print(doubles)

even_numbers = [number for number in numbers if number % 2 == 0]
print(even_numbers)

vowel_animals =[animal.title() for animal in animals if animal[0] in 'aeiou']
print(vowel_animals)
```

```
[2, 10, 4, 24, 28, 14, 36]
[2, 12, 14, 18]
['Aardvark', 'Opossum']
```

In [23]:

```
numbers = [1, 5, 2, 12, 14, 7, 18]
# a set comprehension
doubles_set = {2*number for number in numbers}  # {2, 4, 10, 14, 24, 28, 36}
print(doubles_set)
```

```
{2, 4, 36, 10, 14, 24, 28}
```

In [24]:

```
# a dict comprehension which uses the number as the key and the doubled number as the value
doubles_dict = {number: 2*number for number in numbers} # {1: 2, 2: 4, 5: 10, 7: 14, 12: 24
print(doubles_dict)
```

```
{1: 2, 2: 4, 5: 10, 7: 14, 12: 24, 18: 36, 14: 28}
```

In [25]:

```
# a generator comprehension
doubles_generator = (2*number for number in numbers)
print(doubles_generator)

for p in doubles_generator:
    print(p)
```

```
<generator object <genexpr> at 0x0000000005F92AF0>
2
10
4
24
28
14
36
```

In [26]:

```
sum_odd = sum(x for x in numbers if x % 2 == 1)
print(sum_odd)
```

```
13
```

The Break and Continue Statements

- The break statement can be used to exit a loop immediately.
- A continue statement can be used to start the next iteration immediately without execution of the loop body after this continue statement.

In [27]:

```
numbers = [1, 5, 2, 12, 14, 7, 18]
for p in numbers:
    if p % 2 == 0:
        continue
    print(p)
```

```
1
5
7
```

In [28]:

```
for p in numbers:
    if p % 2 == 0:
        break
    print(p)
```

```
1
5
```

In [29]:

```
for p in numbers:
    if p % 2 == 0:
        continue
    print(p)
else:
    print('exit loop normally')
```

```
1
5
7
exit loop normally
```

In [30]:

```
for p in numbers:
    if p % 2 == 0:
        break
    print(p)
else:
    print('exit loop normally')
```

```
1
5
```

Using Loops to Simplify Codes

A straightforward code for getting personal information may be as follows:

```
name = input("Please enter your name: ")
surname = input("Please enter your surname: ")
age = int(input("Please enter your age: "))
height = float(input("Please enter your height: "))
weight = float(input("Please enter your weight: "))
```

In Python, we can do the same thing in the following manner:

```
person = {}
for prop,cast in [("name",str), ("surname",str), ("age",int), ("height",float),
 ("weight",float)]:
    person[prop] = cast(input("Please enter your {}: ".format(prop)))
```

Read Input Until EOF

Read stdin until EOF

```
import sys

for line in sys.stdin:
    # process each line
```

Read input file until EOF

```
with open(filename,'r') as fp:
    for line in fp:
        # process each line
```

In [31]:

```
lines = []
with open('myfile.txt','r') as infp:
    #for line in infp:
    #   process each line
    lines = [line for line in infp]
print(lines)
```

```
['0\n', '1\n', '2\n', '3\n', '4\n', '5\n', '6\n', '7\n', '8\n', '9\n', '10
\n', '11\n', '12\n', '13\n', '14\n', '15\n', '16\n', '17\n', '18\n', '19\n',
'20\n', '21\n', '22\n', '23\n', '24\n', '25\n', '26\n', '27\n', '28\n', '29
\n', '30\n', '31\n', '32\n', '33\n', '34\n', '35\n', '36\n', '37\n', '38\n',
'39\n', '40\n', '41\n', '42\n', '43\n', '44\n', '45\n', '46\n', '47\n', '48
\n', '49\n', '50\n', '51\n', '52\n', '53\n', '54\n', '55\n', '56\n', '57\n',
'58\n', '59\n', '60\n', '61\n', '62\n', '63\n', '64\n', '65\n', '66\n', '67
\n', '68\n', '69\n', '70\n', '71\n', '72\n', '73\n', '74\n', '75\n', '76\n',
'77\n', '78\n', '79\n', '80\n', '81\n', '82\n', '83\n', '84\n', '85\n', '86
\n', '87\n', '88\n', '89\n', '90\n', '91\n', '92\n', '93\n', '94\n', '95\n',
'96\n', '97\n', '98\n', '99\n']
```

In [ ]:

```
```