Pierian-Data / Complete-Python-3-Bootcamp Public

Code Issues 92 Pull requests 74 Actions Projects Security Insights

43b8ab8f1c Complete-Python-3-Bootcamp / 02-Python Statements / 05-Useful-Operators.ipynb

Go to file ...

Pierian-Data resetting for 2020 updates Latest commit e6583ca on Jun 1, 2020 History

1 contributor

636 lines (636 sloc) | 12.8 KB

Raw Blame

Useful Operators

There are a few built-in functions and "operators" in Python that don't fit well into any category, so we will go over them in this lecture, let's begin!

range

The range function allows you to quickly *generate* a list of integers, this comes in handy a lot, so take note of how to use it! There are 3 parameters you can pass, a start, a stop, and a step size. Let's see some examples:

In [1]:
```
range(0,11)
```

Out[1]:
```
range(0, 11)
```

Note that this is a **generator** function, so to actually get a list out of it, we need to cast it to a list with **list()**. What is a generator? Its a special type of function that will generate information and not need to save it to memory. We haven't talked about functions or generators yet, so just keep this in your notes for now, we will discuss this in much more detail in later on in your training!

In [3]:
```
# Notice how 11 is not included, up to but not including 11, just like slice notation!
list(range(0,11))
```

Out[3]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [4]:
```
list(range(0,12))
```

Out[4]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

In [6]:
```
# Third parameter is step size!
# step size just means how big of a jump/leap/step you
# take from the starting number to get to the next number.

list(range(0,11,2))
```

Out[6]:
```
[0, 2, 4, 6, 8, 10]
```

In [7]:
```
list(range(0,101,10))
```

Out[7]:
```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

enumerate

enumerate is a very useful function to use with for loops. Let's imagine the following situation:

In [8]:
```
index_count = 0

for letter in 'abcde':
    print("At index {} the letter is {}".format(index_count,letter))
    index_count += 1
```

```
At index 0 the letter is a
At index 1 the letter is b
At index 2 the letter is c
At index 3 the letter is d
At index 4 the letter is e
```

Keeping track of how many loops you've gone through is so common, that enumerate was created so you don't need to worry about creating and updating this index_count or loop_count variable

In [10]:

```
# Notice the tuple unpacking!

for i,letter in enumerate('abcde'):
    print("At index {} the letter is {}".format(i,letter))
```

```
At index 0 the letter is a
At index 1 the letter is b
At index 2 the letter is c
At index 3 the letter is d
At index 4 the letter is e
```

zip

Notice the format enumerate actually returns, let's take a look by transforming it to a list()

In [12]:

```
list(enumerate('abcde'))
```

Out[12]:

```
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
```

It was a list of tuples, meaning we could use tuple unpacking during our for loop. This data structure is actually very common in Python , especially when working with outside libraries. You can use the **zip()** function to quickly create a list of tuples by "zipping" up together two lists.

In [13]:

```
mylist1 = [1,2,3,4,5]
mylist2 = ['a','b','c','d','e']
```

In [15]:

```
# This one is also a generator! We will explain this later, but for now let's transform it to a list
zip(mylist1,mylist2)
```

Out[15]:

In [17]:

```
list(zip(mylist1,mylist2))
```

Out[17]:

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

To use the generator, we could just use a for loop

In [20]:

```
for item1, item2 in zip(mylist1,mylist2):
    print('For this tuple, first item was {} and second item was {}'.format(item1,item2))
```

```
For this tuple, first item was 1 and second item was a
For this tuple, first item was 2 and second item was b
For this tuple, first item was 3 and second item was c
For this tuple, first item was 4 and second item was d
For this tuple, first item was 5 and second item was e
```

in operator

We've already seen the **in** keyword during the for loop, but we can also use it to quickly check if an object is in a list

```
In [21]: 'x' in ['x','y','z']
```

```
Out[21]: True
```

```
In [22]: 'x' in [1,2,3]
```

```
Out[22]: False
```

not in

We can combine **in** with a **not** operator, to check if some object or variable is not present in a list.

```
In [1]: 'x' not in ['x','y','z']
```

```
Out[1]: False
```

```
In [2]: 'x' not in [1,2,3]
```

```
Out[2]: True
```

min and max

Quickly check the minimum or maximum of a list with these functions.

```
In [26]: mylist = [10,20,30,40,100]
```

```
In [27]: min(mylist)
```

```
Out[27]: 10
```

```
In [44]: max(mylist)
```

```
Out[44]: 100
```

random

Python comes with a built in random library. There are a lot of functions included in this random library, so we will only show you two useful functions for now.

```
In [29]: from random import shuffle
```

```
In [35]: # This shuffles the list "in-place" meaning it won't return
         # anything, instead it will effect the list passed
         shuffle(mylist)
```

```
In [36]: mylist

Out[36]: [40, 10, 100, 30, 20]
```

```
In [39]: from random import randint
```

```
In [41]: # Return random integer in range [a, b], including both end points.
         randint(0,100)

Out[41]: 25
```

```
In [42]: # Return random integer in range [a, b], including both end points.
         randint(0,100)

Out[42]: 91
```

input

```
In [43]: input('Enter Something into this box: ')

         Enter Something into this box: great job!
Out[43]: 'great job!'
```