KOLT Python
Lists, Strings & Loops

İpek Köprülülü

Monday 4th March, 2019

KOLT

Agenda

1. Recap
2. Lists
3. Strings
4. Loops

`bool` Operators

How to represent logical operations in Python? (and, or, not)

A	B	A or B	A and B	not A
True	True	True	True	False
True	False	True	False	False
False	True	True	False	True
False	False	False	False	True

`True or False and False` ⇒ **`True`**

- **`and`**
- **`or`**
- **`not`**

WHY?

Short-Circuit Evaluation

`X`: Any boolean value

`True or X` ⇒ **`True`**

`False and X` ⇒ **`False`**

Python is smart enough to take advantage of this!

```
1/0  # => ZeroDivisionError
True or 1/0 # => True
False and 1/0 # => False
1/0 or True # => ZeroDivisionError
1/0 and False # => ZeroDivisionError
```

Arithmetic Operators

These operations are applicable on Numeric types: `int` and `float`

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division
- `//`: Floor(integer) Division
- `%`: Modulo
- `**`: Power

```
3.2 + 1.4   # => 4.6
3.2 - 1   # => 2.2
3.2 * 1.2   # => 3.84
3.5 / 1.5   # => 2.3333333333333335
3.5 // 1.5  # => 2.0
3.5 % 1.5   # => 0.5
2 ** 10   # => 1024
```

Comparison Operators

- <: Strictly less than
- <=: Less than or equal
- >: Strictly greater than
- >=: Greater than or equal
- ==: Equal
- !=: Not equal

```
3.0 == 3   # => True
3.0 >= 3   # => True
# Small-case characters
# have bigger ASCII value
'Aa' > 'aa' # => False
'hi' == 'hi'   # => True
'a' == None   # => True
3 > 'a'   # => TypeError
3 == 'a'   # => False
```

Assignment Operators

Operator	Usage	Equivalent
`+=`	`val += 3`	`val = val + 3`
`-=`	`val -= 3`	`val = val - 3`
`*=`	`val *= 3`	`val = val * 3`
`/=`	`val /= 3`	`val = val / 3`
`%=`	`val %= 3`	`val = val % 3`
`**=`	`val **= 3`	`val = val ** 3`
`//=`	`val //= 3`	`val = val // 3`

Branching

```
if <condition>:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

```
if <condition>:
    <expression>
    <expression>
    ...
elif <condition>:
    <expression>
    <expression>
    ...
...
else:
    <expression>
    <expression>
    ...
```

- `<condition>` has a **`bool`** value (`True` or `False`)
- Which expressions will be evaluated in which conditions?

Lists

myList = [1, 2, 3]

```
empty_list = []
letters = ['a', 'b', 'c', 'd']
numbers = [2, 3, 5]
```

```
mixed_list = [4, 13, 'hello']
```

Appending

Append elements at the end of a list by **append()**

```
numbers = [1, 2, 3]
numbers.append(7) # => numbers = [1, 2, 3, 7]
numbers.append(11) # => numbers = [1, 2, 3, 7, 11]

a_list = [1, 'a', 'python', 4.2]
a_list.append(3) # => a_list = [1, 'a', 'python', 4.2, 3]
a_list.append('hello') # => a_list = [1, 'a', 'python', 4.2, 3, 'hello']
```

```
x = [1, 2, 3]
y = [4, 5]
x.append(y) # => x = [1, 2, 3, [4, 5]]
```

Removing An Element

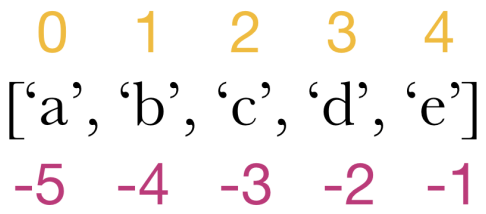
Remove elements in a list by **remove()**

```
x = [1, 2, 3, 4]

x.remove(2) # => x = [1, 3, 4]

y = ['a', 'b', 'c']

y.remove('b') # => y = ['a', 'c']
```

```
x = [1, 2, 5, 4, 2, 6]

x.remove(2) # => x = [1, 5, 4, 2, 6]
```

Inspecting List Elements

Access elements at a particular index




```
x = [1, 2, 'a', 'hello']

x[0] # => 1
x[1] # => 2
x[2] # => 'a'
x[3] # => 'hello'


x[-1] # => 'hello'
x[-2] # => 'a'
x[-3] # => 2
x[-4] # => 1
```

Inspecting List Elements

Slice lists by using **`[start:stop:step]`**

```
x = [1, 2, 3, 4, 5]

x[2:4] # => [3,4]
x[3:4] # => [4]
x[1:-1] # => [2,3,4]
```

```
y = ['a', 'b', 'c', 'd', 'e', 'f']

y[:3] # => ['a', 'b', 'c']
y[2:] # => ['c', 'd', 'e', 'f']
y[:-1] # => ['a', 'b', 'c', 'd', 'e']

y[:] # => ['a', 'b', 'c', 'd', 'e', 'f']
```

Inspecting List Elements

```
y = ['a', 'b', 'c', 'd', 'e', 'f']

y[1:5:2] # => ['b', 'd']
y[::3] # => ['a', 'd']
```

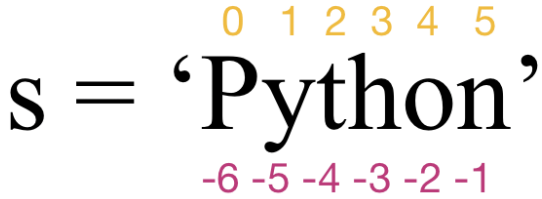
```
y = ['a', 'b', 'c', 'd', 'e', 'f']

y[::-1] # => ['f', 'e', 'd', 'c', 'b', 'a']
```

Nested Lists

Lists can contain lists

```
x = [[15, 4, 20, 7], [3, 18, 9]]

x[1] # => [3, 18, 9]
x[1][2] # => 9
x[0][2:] # => [20, 7]
```

Strings



Indexing & Slicing

```
s = 'Python'

s[1] # => 'y'
s[0:4] # => 'Pyth'
s[:3] # => 'Pyt'
s[3:] # => 'hon'
s[:] # => 'Python'
```

```
s = 'Python'

s[:5:2] # => 'Pto'
s[1:4:3] # => 'y'
s[::3] # => 'Ph'
s[::-1] # => 'nohtyP'
```

in Operator

Search an operand in the specified sequence by using **in**

```
0 in [] # => False
'y' in  'Python' # => True
23 in ['hello', 40, 'a', 5] # => False
23 in ['hello', 40, 'a', 23] # => True
23 in ['hello', 40, 'a', '23'] # => False
```

- Works with lists and strings
- Works with ranges

range() Function

`range(`**`start, stop, step`**`)` is a function to create ranges

```
a = range(3) # => generates 0, 1, 2
b = range(0,3) # => generates 0, 1, 2
c = range(2,4) # => generates 2, 3
d = range(0,6,2) # => generates 0, 2, 4

0 in a # => True
1 in b # => True
4 in c # => False
2 in d # => True
6 in d # => False
```

len() Function

`len()` is an operator to determine the size of lists, strings, etc.

```
s = 'Python'
len(s) # => 6

my_list = [0, 1, 2, 3]
len(my_list) # => 4
```

For Loops

```
for <item> in <iterable>:
    <expression>
    <expression>
    ...
```

```
for ch in 'Python':
    print(ch)
```

```
for num in [4,23,12,0,50]:
    print(num * 3,sep=".")
```

```
for i in range(0,8):
    print(i)
```

While Loops

```
while <condition>:
    <expression>
    <expression>
    ...
```

```
x = 15
while x > 10:
    print(x)
    x-=1
```

```
x = 10
a_list = [1, 2, 3, 4, 5, 6, 7]

while len(a_list) < x:
    a_list.append(0)
    print(a_list)
```

Break, Continue & Pass

Break terminates the closest for or while loop

```
for i in range(0, 5):
    if i % 2 == 1:
        break
    print(i)
```

```
x = 1
while x < 100:
    x *= 2
    if (x+1) % 3 == 0:
        break
    print(x)
```

Continue continues with the next iteration of the loop

```
for i in range(0, 5):
    if i % 2 == 1:
        continue
    print(i)
```

```
x = 1
while x < 100:
    x *= 2
    if (x+1) % 3 == 0:
        continue
    print(x)
```

Break, Continue & Pass

Pass does not have an effect

```
for letter in 'Python':
   if letter == 'y':
      pass
      print ('In pass case')
   print(letter)
```

- Loops, conditional statements, functions etc. cannot be empty
- Use when you have to create one