KOLT Python
Functions

Ahmet Uysal

Monday 21st October, 2019

KOLT

Agenda

1. Recap

2. Functions

- Defining Functions
- return Statement
- Parameters

While Loops

Repeat some `<expression>`s as long as a `<condition>` is `True`.

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

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

```
counter = 11
while counter > 6:
    counter -= 1
    print(2**counter)
    counter -= 1
```

`<condition>` is only checked **before** each execution.

List Slicing

Access collection of elements with **`[start:stop:step]`**
Gives a list, even when number of elements is not bigger than 1.

```
numbers[0::2]   # => [0, 2, 4]
numbers[:]  # => [0, 1, 2, 3, 4, 5]
numbers[1:]   # => [1, 2, 3, 4, 5]
numbers[-2:]   # => [4, 5]
numbers[1:4]   # => [1, 2, 3]
numbers[1:1]   # => []
numbers[-99:99]   # => [0, 1, 2, 3, 4, 5]
numbers[::-1]   # => [5, 4, 3, 2, 1, 0]
numbers[::-2]   # => [5, 3, 1]
```

Slices with `step` = 1 are called **Basic Slice**.
Slices with `step != 1` are called **Extended Slice**.

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
```

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
```

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)
```

Break, Continue & Pass

break immidiately terminates the closest 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 skips to 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

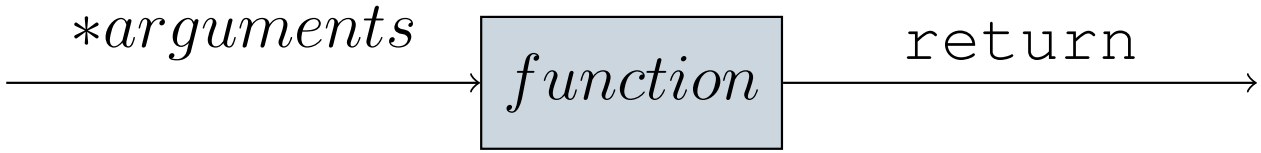
```
for letter in 'Python':
    if letter == 'y':
        pass
    else:
        print(letter)
```

- Loops, conditional statements, functions etc. cannot be empty

Functions

Functions are blocks of **organized**, **reusable** code that carry some **specific** tasks.

- **`input([prompt])`:**
  If the prompt **argument** is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and **returns** that. When `EOF` is read, `EOFError` is **raised**.



Defining Functions

`def` keyword introduces a function *definition*.

```
def function_name():
    <expression>
    <expression>
    ...
```

```
def function_name(parameter1, parameter2, ...):
    <expression>
    <expression>
    ...
```

```
def function_name(parameter1, parameter2, ...):
    <expression>
    ...
    return value
```

Functions

```
def input_float(prompt):
    """Takes and returns a float value from user."""
    return float(input(prompt))
```

```
def fibonacci_series(limit):
    """Returns a list of the Fibonacci series up to limit."""
    fib_list = []
    first = 0
    second = 1
    while first < limit:
        fib_list.append(first)
        first, second = second, first + second
    return fib_list

print(fibonacci_series)
```

Functions

Defining a `function` only makes it available.
You should *call* the `function` to execute.

```
fib_100 = fibonacci_series(100)
what_is_going_on = print(fib_100)
```

return Statement

```
def double(a):
    return a*2
    print("Doubled")

num = double(4)
print(num)
```

Return **immidiately** terminates the function. So, the output is 8.

return Statement

Every function returns **one** value!

```
value = print('Hello, World!')
print(value)
```

Functions implicitly return `None` if they complete without a return statement.

Default Parameters

The values of parameters can be set to used as default.
In **`print(*args, sep=' ', end='\n')`**, sep and end has default values.

```
def info(num, name='NoInfo', surname='NoInfo', ID='NoInfo'):
    print(num, name, surname, ID)
```

Valid Uses

```
# 1 positional argument
info(2)
# 2 positional arguments
info(2, 'Jane')
# 3 positional arguments
info(2, 'Jane', 'Doe')
# 4 positional arguments
info(2, 'Jane', 'Doe', 20)
```

```
# 1 keyword argument
info(num=1)
# 2 keyword arguments
info(name='Jane', num=9)
# 2 keyword arguments
info(num=9, name='Jane')
# 1 positional, 1 keyword
info(2, 'John', ID=13)
```

Default Parameters

```
def info(num, name='NoInfo', surname='NoInfo', ID='NoInfo'):
    print(num, name, surname, ID)
```

Invalid Usages

```
# required argument missing
info()
# non-keyword argument after a keyword arg
info(num=2, 'Jane')
# duplicate value for the same argument
info(2, num=3)
# unknown keyword argument
info(person='Jane')
```

Variadic Positional Arguments

How to allow function to accept arbitrary number of arguments.
In `print(*args, sep=' ', end='\n')`, you can put as many args as you want.

Suppose we want a `max` function that works as so:
max(3, 5) gives 5.
max(3, 4, 2) gives 4.
product(3, 5, -1, 2, 10, 20, 13, 34) gives 34.

Variadic Positional Arguments: my_max

```
def my_max(*nums):
    """Returns the maximum of the given arguments.
    Returns -infinity if no arguments are given."""
    max_num = -float('inf')
    for n in nums:
        if n > max_num:
            max_num = n
    return max_num
```