

KOLT Python Functions

İpek Köprülülü

Monday 11th March, 2019

KOLT

1. Recap

○○○○○○○○○○

2. Functions

○○○○○○○○○○

Agenda

1. Recap

2. Functions



Lists

- Group values together. `my_values = [1, 'a', None]`
- You can think of each element as a variable, accessed by **indexing**
- You can do everything you do to variables to list elements:
 - Assign new values: `my_values[0] = 3`
 - Use shorthand assignment operators: `my_values[1] += 'bc'`
 - Learn their type: `type(my_values[2]) # => <class 'NoneType'>`
 - Change their type: `my_values[2] = True`
 - Compare their value: `if my_values[0] == my_values[1]: ...`
- What happens when we call `my_values[3] = 3`? # => **IndexError**

List Indexing

Access elements at a particular index

0	1	2	3	4
['a',	'b',	'c',	'd',	'e']
-5	-4	-3	-2	-1

```
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
```

List Slicing

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

```
numbers = [0, 1, 2, 3, 4, 5]
"""
ASCII art analogy :)
+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | => Indices
+---+---+---+---+---+---+
 0   1   2   3   4   5   6 => Borders
-6  -5  -4  -3  -2  -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**.

List Mutation

`list.append(x)`: Append x to end of the sequence

`list.insert(i, x)`: Insert x to index i

`list.pop(i=-1)`: Remove and return element at index i

`list.remove(x)`: Remove first occurrence of x

`list.extend(iterable)`: Add all elements in iterable to end of list

`list[i] = new_value`: Update value of index i with new value

`list[basic_slice] = iterable`: Change elements in basic slice with elements in iterable, sizes can be different: `numbers[:]` = []

`list[extended_slice] = iterable`: Change elements in extended slice with elements in iterable 1-1, sizes must be equal.

Some Other List Operations

in operator: Check whether an element is in list. `3 in numbers` \Rightarrow `True`

len(list): Returns the length of list(and other collections).

list.index(value, start=0, stop=len(list)): Return first index of value.

list.count(value): Count number of occurrences of value in list.

list.reverse(): Reverse the list (in-place)

list.sort(): Sort list elements (in-place)

For more, type `help(list)` in your interactive interpreter.

Strings

Special kind of **lists**! `name = 'Python'`

You can do:

- Indexing: `name[2] ⇒ 't'`
- Slicing: `name[::-1] ⇒ 'nohtyP'`
- Search by `in` operator: `'yt' in name ⇒ True`

You can not do:

- String mutation: `name[2]='H' ⇒ TypeError`

Special functions about strings: `str.isnumeric()`,
`str.capitalize()`, `str.format(...)`, `str.find()` ...

Loops

Do something for many elements or based on a condition.

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

Similar to simple if blocks, but runs again and again until condition check fails.

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

Iterable: collection of **ordered** elements.
What is next after this item?

For Loops

What is next after this item?

`numbers[1]` is after `numbers[0]` \neq **`numbers[1] > numbers[0]`**

Examples of iterables: lists, strings, ranges

Ranges

`range(start, stop, step)`: creates a sequence of integers from start (inclusive) to stop (exclusive) by step.

Can be **indexed** and **sliced**

`len()` and `in` operator can be used

For Loops

```
names = ['Mario', 'Peter', 'Anna', 'Paul', 'Anna']

for number in range(2, 5):
    # In every iteration, we have a different value from iterable
    # We can access the value with the name we specified
    print(number)
    # range is collection of integers, we can use ints in indexing
    print('Hello', names[number])

# Nested loops
for name in names:
    # In every iteration name changes, in the order of names
    if name != names[number]:
        print(name, 'says hello to', names[number])
```

Break, Continue

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)
```

For Else, While Else

else in branching: executed when all of the conditions in upper if/elif blocks are False

else in loops: executed when loop is terminated **without** a break statement

```
while <condition>:
    <expression>
    if <condition>:
        break
    <expression>
    ...
# This block is executed if
# while loop is not terminated by break
# Note: this block runs even when
# condition is False at initial evaluation
else:
    <expression>
    <expression>
```

```
for item in iterable:
    <expression>
    if <condition>:
        break
    <expression>
    ...
# This block is executed if
# for loop is not terminated by break,
# it iterated all elements
else:
    <expression>
    <expression>
```

Functions

Functions are

- pieces of codes written to carry out some specified tasks.
- used to bundle a set of instructions that you want to use repeatedly.
- block of codes which only call when needed to avoid complexity.
- The def keyword is used to define a new function.

Functions

Functions are

- pieces of codes written to carry out some specified tasks.
- used to bundle a set of instructions that you want to use repeatedly.
- block of codes which only call when needed to avoid complexity.
- The def keyword is used to define a new function.

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

Functions

Functions are

- pieces of codes written to carry out some specified tasks.
- used to bundle a set of instructions that you want to use repeatedly.
- block of codes which only call when needed to avoid complexity.
- The def keyword is used to define a new function.

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

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


Functions

Functions are

- pieces of codes written to carry out some specified tasks.
- used to bundle a set of instructions that you want to use repeatedly.
- block of codes which only call when needed to avoid complexity.
- The def keyword is used to define a new function.

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

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

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

Functions

```
def sayHello():  
    print("Hello")  
  
sayHello() # => Hello
```

Functions

```
def sayHello():  
    print("Hello")
```

```
sayHello() # => Hello
```

```
def getANumber():  
    num = int(input("Enter a number: "))  
    print("Your number is", num)
```

```
getANumber()
```

```
# Enter a number: 10
```

```
# Your number is 10
```

Functions

```
def sum(a, b, c):  
    print(a+b+c)  
  
sum(1, 2, 3) # => 6  
sum(2, 5, 6) # => 13  
sum(0, 0, 0) # => 0
```

You should call the function in your code to make it work.

Functions

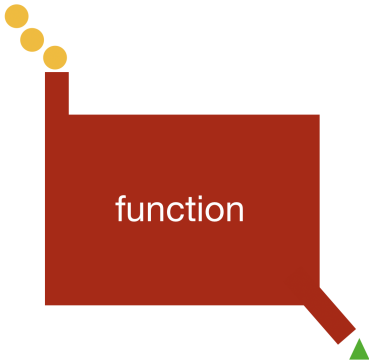
```
def sum(a, b, c):  
    print(a+b+c)  
  
sum(1, 2, 3) # => 6  
sum(2, 5, 6) # => 13  
sum(0, 0, 0) # => 0
```

You should call the function in your code to make it work.

```
def factorial(n):  
    result = 1  
    if n == 0 or n == 1:  
        print(1)  
    else:  
        for i in range(1, n+1):  
            result *= i  
        print(result)  
  
factorial(0) # => 1  
factorial(1) # => 1  
factorial(3) # => 6  
factorial(4) # => 24  
factorial(5) # => 120
```

Return

All functions return some value even if that value is None.



Return

```
def factorial(n):  
    result = 1  
    if n == 0 or n == 1:  
        return 1  
  
    for i in range(1,n+1):  
        result *= i  
    return result
```

You should call the function and assign it to a variable to hold the value.

```
a = factorial(0)  
b = factorial(1)  
c = factorial(3)  
d = factorial(4)  
e = factorial(5)  
  
print(a) # => 1  
print(b) # => 1  
print(c) # => 6  
print(d) # => 24  
print(e) # => 120
```

Return

```
def sum(a, b, c):  
    return a+b+c  
  
num = sum(1, 2, 3)  
print(num) # => 6
```


Return

```
def sum(a, b, c):  
    return a+b+c
```

```
num = sum(1, 2, 3)  
print(num) # => 6
```

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

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

Return

```
def sum(a, b, c):  
    return a+b+c
```

```
num = sum(1, 2, 3)  
print(num) # => 6
```

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

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

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

Default Parameters

The values of parameters can be set to used as default.

In `print(*args, sep=' ', end='\n')`, `sep` and `end` are defined as default parameters.

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

Default Parameters

The values of parameters can be set to used as default.

In `print(*args, sep=' ', end='\n')`, `sep` and `end` are defined as default parameters.

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

```
# 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

The values of parameters can be set to used as default.

In `print(*args, sep=' ', end='\n')`, `sep` and `end` are defined as default parameters.

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

Default Parameters

The values of parameters can be set to used as default.

In `print(*args, sep=' ', end='\n')`, `sep` and `end` are defined as default parameters.

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

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

Variadic Positional Arguments

It is used to let the function accept any number of arguments.

In `print(*args, sep=' ', end='\n')`, you can put as many args as you want.

Suppose we want a product function that works as so:

`product(3, 5)` gives 15.

`product(3, 4, 2)` gives 24.

`product(3, 5, scale=10)` gives 150.

Variadic Positional Arguments

It is used to let the function accept any number of arguments.

In `print(*args, sep=' ', end='\n')`, you can put as many args as you want.

Suppose we want a product function that works as so:

`product(3, 5)` gives 15.

`product(3, 4, 2)` gives 24.

`product(3, 5, scale=10)` gives 150.

```
def product(*nums, scale = 1):  
    p = scale  
    for n in nums:  
        p *= n  
    return p
```


Local & Global Variables

- Local variables are created in functions.
- Global variables are created out of the functions.

```
x = 10 # => global

def func():
    x = 5 # => local
    y = 7 # => local
    print(x, y)

func()
print(x)
```

Local & Global Variables

- Local variables are created in functions.
- Global variables are created out of the functions.

```
x = 10 # => global

def func():
    x = 5 # => local
    y = 7 # => local
    print(x, y)

func()
print(x)
```

```
x = 10

def func():
    print(x)

func() # => 10
```

Local & Global Variables

- Local variables are created in functions.
- Global variables are created out of the functions.

```
x = 10 # => global

def func():
    x = 5 # => local
    y = 7 # => local
    print(x, y)

func()
print(x)
```

```
x = 10

def func():
    print(x)

func() # => 10
```

```
def func():
    a = 2
    print(a)

func()
print(a) # => not defined (gives error)
```

Local & Global Variables

```
x = 2

def func(num):
    num = 6
    print(num)

func(x)
print(x)
```

Local & Global Variables

```
x = 2

def func(num):
    num = 6
    print(num)

func(x)
print(x)
```

Prints

6

2

Local & Global Variables

```
x = 2

def func(num):
    num = 6
    print(num)
```

```
func(x)
print(x)
```

```
x = 2

def func():
    x = 6
    print(x)
```

```
func()
print(x)
```

Prints

6

2

Local & Global Variables

```
x = 2

def func(num):
    num = 6
    print(num)

func(x)
print(x)
```

Prints

6
2

```
x = 2

def func():
    x = 6
    print(x)

func()
print(x)
```

Prints

6
2

Local & Global Variables

```
x = 2

def func(num):
    num = 6
    print(num)

func(x)
print(x)
```

Prints

6
2

```
x = 2

def func():
    x = 6
    print(x)

func()
print(x)
```

Prints

6
2

```
x = 2

def func():
    global x
    x = 6
    print(x)

func()
print(x)
```


Local & Global Variables

```
x = 2

def func(num):
    num = 6
    print(num)

func(x)
print(x)
```

Prints

6
2

```
x = 2

def func():
    x = 6
    print(x)

func()
print(x)
```

Prints

6
2

```
x = 2

def func():
    global x
    x = 6
    print(x)

func()
print(x)
```

Prints

6
6

Lambda

We can write short functions in one line by using `lambda`.

```
function_name = lambda parameter1, parameter2, ... : return_value
```

Lambda

We can write short functions in one line by using `lambda`.

```
function_name = lambda parameter1, parameter2, ... : return_value
```

```
def double(x):  
    return x*2
```

Lambda

We can write short functions in one line by using `lambda`.

```
function_name = lambda parameter1, parameter2, ... : return_value
```

```
def double(x):  
    return x*2
```

```
double = lambda x : x*2
```

Lambda

We can write short functions in one line by using `lambda`.

```
function_name = lambda parameter1, parameter2, ... : return_value
```

```
def double(x):  
    return x*2
```

```
def sumAndPrint(x, y, z):  
    print(x+y+z)
```

```
double = lambda x : x*2
```

Lambda

We can write short functions in one line by using `lambda`.

```
function_name = lambda parameter1, parameter2, ... : return_value
```

```
def double(x):  
    return x*2
```

```
def sumAndPrint(x,y,z):  
    print(x+y+z)
```

```
double = lambda x : x*2
```

```
sumAndPrint = lambda x,y,z : print(x+y+z)
```

Lambda

We can write short functions in one line by using `lambda`.

```
function_name = lambda parameter1, parameter2, ... : return_value
```

```
def double(x):  
    return x*2
```

```
def sumAndPrint(x,y,z):  
    print(x+y+z)
```

```
def reverseString(s):  
    return s[::-1]
```

```
double = lambda x : x*2
```

```
sumAndPrint = lambda x,y,z : print(x+y+z)
```

Lambda

We can write short functions in one line by using **lambda**.

```
function_name = lambda parameter1, parameter2, ... : return_value
```

```
def double(x):  
    return x*2
```

```
def sumAndPrint(x,y,z):  
    print(x+y+z)
```

```
def reverseString(s):  
    return s[::-1]
```

```
double = lambda x : x*2
```

```
sumAndPrint = lambda x,y,z : print(x+y+z)
```

```
reverseString = lambda s: s[::-1]
```