

1. Recap  
oooooooooooo

2. Lists(Cont.)  
oooooo

3. For Loops  
ooooo

4. Functions  
oooooooooooo

# KOLT Python

## Lists, For Loops & Functions

Ahmet Uysal

Monday 14<sup>th</sup> October, 2019

# KOLT



1. Recap  
oooooooooooo

2. Lists(Cont.)  
oooooo

3. For Loops  
ooooo

4. Functions  
oooooooooooo

# Agenda

1. Recap

2. Lists(Cont.)

3. For Loops

4. Functions

# Strings

```
my_string = 'abcde'
```

```
0 1 2 3 4  
'a b c d e'  
-5 -4 -3 -2 -1
```

```
print(my_string[2]) ⇒ prints c
```

```
print(my_string[-2]) ⇒ prints d
```

## Indexing & Slicing

Access specific characters using **indexing**, i.e, `[index]`  
**Slice** strings by using `[start:stop:step]`

```
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'
```



## String Operations

```
print('This a simple calculator program.')
number1 = input('Please enter the first number:')
number2 = input('Please enter the second number:')
print(f'{number1}+{number2} is {number1 + number2}')
```

```
number1 = int(input('First number:'))
number2 = input('Please enter the second number:')
print(f'{number1}x{number2} is {number1 * number2}')
```

**str1 + str2** ⇒ **Concatenate** str1 and str2

**str1 \* n** ⇒ Repeat str1 *n* times.



## 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.

# Lists



Imagine variables, but with limitless capacity. . .

```
sunnyside = ['Mr. Potato Head', 'Hamm',  
'Buzz Lightyear', 'Slinky Dog']
```

## Lists

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

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



## Accessing Elements

```
values = [1, 'hello', None, [3], True]
```

0	1	2	3	4	
[	1,	'hello',	None,	[3],	True]
-5	-4	-3	-2	-1	

Use **indexing** to access and **update** elements inside list.

```
print(values[2])  
values[2] = 'new value'
```

## Adding New Elements

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]]
```

# 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']
```



# Removing An Element

# Removing An Element

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

## Removing An Element

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

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

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

numbers_repeated = [1, 2, 5, 4, 2, 6]
numbers_repeated.remove(2) # => number_repeated = [1, 5, 4, 2, 6]

my_list = [1, 'a']
my_list.remove('b') # => ValueError
```

## Removing An Element

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

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

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

numbers_repeated = [1, 2, 5, 4, 2, 6]
numbers_repeated.remove(2) # => number_repeated = [1, 5, 4, 2, 6]

my_list = [1, 'a']
my_list.remove('b') # => ValueError
```

How to avoid ValueError?



## Removing An Element

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

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

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

numbers_repeated = [1, 2, 5, 4, 2, 6]
numbers_repeated.remove(2) # => number_repeated = [1, 5, 4, 2, 6]

my_list = [1, 'a']
my_list.remove('b') # => ValueError
```

How to avoid ValueError? (Hint: **Branching**)

# in Operator

## in Operator

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

## 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
```

## 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 both lists and strings

## 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 both lists and strings
- Works with ranges

# len() Function

## len() Function

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



## 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
```

## 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**.

## 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.

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

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

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



# range() Function

## 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 Loops

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



# For Loops

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

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

# 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 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)
```

## Example: Mail Sender

## Example: Mail Sender

Fill out the attendance form: [tiny.cc/kolt-python](https://tiny.cc/kolt-python)



## Break, Continue & Pass

**break** immediately terminates the closest loop

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

## Break, Continue & Pass

**break** immediately 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)
```



## Break, Continue & Pass

**break** immediately 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)
```



## Break, Continue & Pass

**break** immediately 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

## Break, Continue & Pass

**pass** does not have an effect

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

## 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

1. Recap

○○○○○○○○○○

2. Lists(Cont.)

○○○○○○

3. For Loops

○○○○○

4. Functions

●○○○○○○○○

# Functions



# Functions

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

## Functions

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

- `input ( [prompt] ) :`

# 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.



## Functions

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

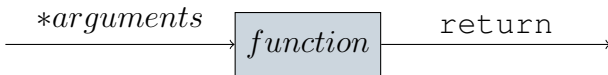
- `input ( [prompt] ) :`  
If the prompt **argument** is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and **returns** that. When EOF is read, `EOFError` is **raised**.

# 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

# Defining Functions

`def` keyword introduces a function *definition*.

# Defining Functions

**def** keyword introduces a function *definition*.

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

# Defining Functions

**def** keyword introduces a function *definition*.

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

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

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



# Functions

```
def inputFloat(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.

# Functions

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

# Functions

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

```
fib_100 = fibonacci_series(100)
```

# 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)
```

# 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)
```

**Every** function returns **one** value!

## 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)
```

**Every** function returns **one** value! Functions implicitly return `None` if they complete without a return statement.

1. Recap

oooooooooooo

2. Lists(Cont.)

oooooo

3. For Loops

ooooo

4. Functions

oooo●oooo

# Return





## Return

```
def double(a):  
    return a*2  
    print("Doubled")  
  
num = double(4)  
print(num)
```

## Return

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

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

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

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

## Return

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

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

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

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

Return **immediatly** terminates the function. So, the output



## 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.

## 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

## 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)
```

## 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',  
        print(num, name, surname, ID))
```



## Default Parameters

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

## Invalid Usages

```
# 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

## 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.

## 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.



## 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) # => NameError
```