KOLT Python

Functions

Ahmet Uysal

Tuesday 25th February, 2020

KOÇ UNIVERSITY

OFFICE OF LEARNING AND TEACHING

Agenda

1. Recap

- Loops
- Lists
- Basic Functions
  - Exercise

2. Functions

- Defining Functions
- return Statement
- Parameters

while & for Loops

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

`while` & `for` Loops

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

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

`while` & `for` Loops

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

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

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

`while` & `for` Loops

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

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

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

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

`while` & `for` Loops

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

Repeat some `<expression>`s as long as a `<condition>` is `True`. `<condition>` is only checked **before** each execution.

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

Repeat some `<expression>`s for **each** element of a `<iterable>`.

`break` & `continue` statements

`break` & `continue` statements can alter the **normal flow** of a loop.

`break` & `continue` statements

`break` & `continue` statements can alter the **normal flow** of a loop.

`break`:

`break` & `continue` statements

`break` & `continue` statements can alter the **normal flow** of a loop.

`break`:

- Immidiately terminates the loop

`break` & `continue` statements

`break` & `continue` statements can alter the **normal flow** of a loop.

`break`:

- Immidiately terminates the loop

`continue`:

`break` & `continue` statements

`break` & `continue` statements can alter the **normal flow** of a loop.

`break`:

- Immidiately terminates the loop

`continue`:

- Jumps to the **next iteration** of the loop

`break` & `continue` statements

`break` & `continue` statements can alter the **normal flow** of a loop.

`break`:

- Immidiately terminates the loop

`continue`:

- Jumps to the **next iteration** of the loop
  - `while`:

`break` & `continue` statements

`break` & `continue` statements can alter the **normal flow** of a loop.

`break`:

- Immidiately terminates the loop

`continue`:

- Jumps to the **next iteration** of the loop
  - `while`: jump to the control step

`break` & `continue` statements

`break` & `continue` statements can alter the **normal flow** of a loop.

`break`:

- Immidiately terminates the loop

`continue`:

- Jumps to the **next iteration** of the loop
  - `while`: jump to the control step
  - `for`:

`break` & `continue` statements

`break` & `continue` statements can alter the **normal flow** of a loop.

`break`:

- Immidiately terminates the loop

`continue`:

- Jumps to the **next iteration** of the loop
  - `while`: jump to the control step
  - `for`: jump to the next element of a the `<iterable>`.

Lists

Imagine variables, but with limitless capacity. . .

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

List Operations

`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.

List Operations (cont.)

`in` operator: Check whether an element is in list.
`3 in numbers` ⇒ `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.

Functions

Functions are blocks of

Functions

Functions are blocks of **organized**,

Functions

Functions are blocks of **organized**, **reusable** code

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.

Defining Functions

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

```
def prepare_base_vegetables():
    print("Chop the tomatoes")
    print("Deseed and slice the peppers")
```

```
def cook():
    print("Cook the vegetables until they soften")
    print("Crack and cook the eggs")
```

Exercise

Assume we have a list that contains scores of all football matches that are played between Fenerbahçe and Galatasaray at Şükrü Saraçoğlu Stadium.

Exercise

Assume we have a list that contains scores of all football matches that are played between Fenerbahçe and Galatasaray at Şükrü Saraçoğlu Stadium.

```
scores = [[5, 1], ..., [1, 3]]
```

Exercise

Assume we have a list that contains scores of all football matches that are played between Fenerbahçe and Galatasaray at Şükrü Saraçoğlu Stadium.

```
scores = [[5, 1], ..., [1, 3]]
```

For both teams, we want to find:

Exercise

Assume we have a list that contains scores of all football matches that are played between Fenerbahçe and Galatasaray at Şükrü Saraçoğlu Stadium.

```
scores = [[5, 1], ..., [1, 3]]
```

For both teams, we want to find:

1. Longest unbeaten runs

Exercise

Assume we have a list that contains scores of all football matches that are played between Fenerbahçe and Galatasaray at Şükrü Saraçoğlu Stadium.

```
scores = [[5, 1], ..., [1, 3]]
```

For both teams, we want to find:

1. Longest unbeaten runs
2. Longest winning streaks

Exercise

Assume we have a list that contains scores of all football matches that are played between Fenerbahçe and Galatasaray at Şükrü Saraçoğlu Stadium.

```
scores = [[5, 1], ..., [1, 3]]
```

For both teams, we want to find:

1. Longest unbeaten runs
2. Longest winning streaks
3. Number of matches to last win

Exercise

Assume we have a list that contains scores of all football matches that are played between Fenerbahçe and Galatasaray at Şükrü Saraçoğlu Stadium.

```
scores = [[5, 1], ..., [1, 3]]
```

For both teams, we want to find:

1. Longest unbeaten runs
2. Longest winning streaks
3. Number of matches to last win

Starter Code

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

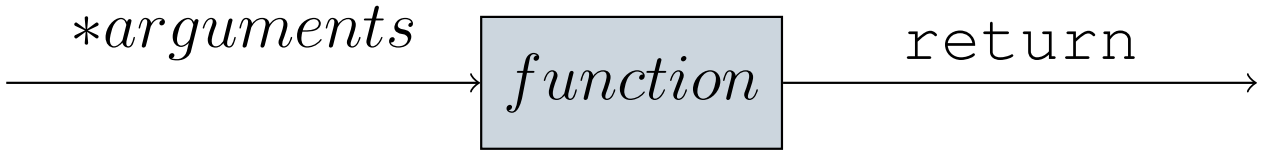
Functions are blocks of **organized**, **reusable** code that carry some **specific** tasks.

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

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.

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)
```

return Statement

return Statement

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

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

return Statement

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

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

Return **immidiately** terminates the function.
So, `print('Doubled')` is not executed by Python.

return Statement

return Statement

Every function returns **one** value!

return Statement

Every function returns **one** value!

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

return Statement

Every function returns **one** value!

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

return Statement

Every function returns **one** value!

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

return Statement

Every function returns **one** value!

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

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

Example Revisited

Example Revisited

```
def largest_unbeaten_run(team_name):
```

Example Revisited

```
def largest_unbeaten_run(team_name):
def largest_winning_streak(team_name):
```

Example Revisited

```
def largest_unbeaten_run(team_name):
def largest_winning_streak(team_name):
def matches_to_last_win(team_name):
```

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')
# 2 positional, 1 keyword
info(2, 'John', ID=13)
```

Default Parameters

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

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')
```

Example Revisited

How can we make our functions return results for Galatasaray by default?

Example Revisited

How can we make our functions return results for Galatasaray by default?

```
def largest_unbeaten_run(team_name
```

Example Revisited

How can we make our functions return results for Galatasaray by default?

```
def largest_unbeaten_run(team_name='GS'):
```

Example Revisited

How can we make our functions return results for Galatasaray by default?

```
def largest_unbeaten_run(team_name='GS'):
def largest_winning_streak(team_name='GS'):
```

Example Revisited

How can we make our functions return results for Galatasaray by default?

```
def largest_unbeaten_run(team_name='GS'):
def largest_winning_streak(team_name='GS'):
def matches_to_last_win(team_name='GS'):
```

Variadic Positional Arguments

Variadic Positional Arguments

Can functions accept arbitrary number of arguments?
In `print(*args, sep=' ', end='\n')`, you can put as many args as you want.

Variadic Positional Arguments

Can functions 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

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
```

Announcements

Fill out the attendance form:
tiny.cc/koltpython
Keyword: **functions**

Announcements

Fill out the attendance form:
tiny.cc/koltpython
Keyword: **functions**
Assignment I: Tic-Tac-Toe is posted!