

1. Recap  
○○○○○○○

2. Logical Operators  
○○○○○

3. Branching  
○○

# KOLT Python

## Basic Operators & Branching

Ceren Kocaoğullar

Monday 30<sup>th</sup> September, 2019

# KOLT



1. Recap  
○○○○○○○

2. Logical Operators  
○○○○○

3. Branching  
○○

# Agenda

1. Recap

2. Logical Operators

3. Branching



## Comments

```
# Single line comments start with a '#'  
  
"""  
Multiline comments can be written between  
three "s and are often used as function  
and module comments.  
"""  
print('Hello, stranger!')
```

Python will basically ignore comments, they are purely written **for humans!**

## Variables

Type	Explanation	Examples
int	represent <b>integers</b>	3, 4, 17, -10
float	represent <b>real numbers</b>	3.0, 1.11, -109.123123
bool	represent <b>boolean</b> truth values	True, False
str	A sequence of characters.	'Hello', '', '3'
NoneType	special and has one value, None	None

- How to create a variable? `variable_name = value`
- How about types? use `type()`
- Can a variable change type? **Yes!** Just assing a new value with any type.
- What if we if want to convert a value between types, i.e, '2' → 2

## Casting

- `int('2') → 2`
- Any possible reasons for casting? -taking user input  
-reading numbers from a file?
- Can we cast every value to every type? **NO!** try  
`int('hello')`

## Console I/O(Input/Output)

```
print(*args, sep=' ', end='\n')
```

- Can take arbitrary number of arguments
- Separates elements with space by default
- Adds newline character '`\n`' to end by default

```
input([prompt])
```

- Prints the prompt to Console
- Program is paused until user enters something
- **returns an `str` object!**

## Arithmetic Operators

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

## Arithmetic Operators

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

- `+`: Addition



## Arithmetic Operators

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

- `+`: Addition
- `-`: Subtraction

## Arithmetic Operators

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

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication

## Arithmetic Operators

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

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division

## Arithmetic Operators

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

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

## Arithmetic Operators

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

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

## Arithmetic Operators

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

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



## 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 # => 2.3333333333333333
3.5 // 1.5 # => 2.0
3.5 % 1.5  # => 0.5
2 ** 10    # => 1024
```

# Assignment Operators

We have already seen '=':



# Assignment Operators

We have already seen '=': `variable_name = value`

## Assignment Operators

We have already seen '=': `variable_name = value`  
Frequently we will update variables' values based on their **old value**.

## Assignment Operators

We have already seen '=': `variable_name = value`  
Frequently we will update variables' values based on their **old value**.

**Ex:** Increment a number: `num = num + 1`

## Assignment Operators

We have already seen '=': `variable_name = value`  
Frequently we will update variables' values based on their **old value**.

**Ex:** Increment a number: `num = num + 1`

Python has shorthand representations for these updates with arithmetic operators.

## Assignment Operators

We have already seen '=': `variable_name = value`  
Frequently we will update variables' values based on their **old value**.

**Ex:** Increment a number: `num = num + 1`

Python has shorthand representations for these updates with arithmetic operators.

`num += 1` is equivalent to `num = num + 1`

## Assignment Operators

We have already seen '=': `variable_name = value`  
Frequently we will update variables' values based on their **old value**.

**Ex:** Increment a number: `num = num + 1`

Python has shorthand representations for these updates with arithmetic operators.

`num += 1` is equivalent to `num = num + 1`

`result *= 2` is equivalent to `result = result * 2`

## Assignment Operators

Operator	Usage	Equivalent
<b>+=</b>	<code>val += 3</code>	<code>val = val + 3</code>
<b>-=</b>	<code>val -= 3</code>	<code>val = val - 3</code>
<b>*=</b>	<code>val *= 3</code>	<code>val = val * 3</code>
<b>/=</b>	<code>val /= 3</code>	<code>val = val / 3</code>
<b>%=</b>	<code>val %= 3</code>	<code>val = val % 3</code>
<b>**=</b>	<code>val **= 3</code>	<code>val = val ** 3</code>
<b>//=</b>	<code>val //= 3</code>	<code>val = val // 3</code>

## 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



## 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

- **and**

## 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

- and
- or

## 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

- and
- or
- not

## 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  $\Rightarrow$

- and
- or
- not

## 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  $\Rightarrow$  **True**

- and
- or
- not

## 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  $\Rightarrow$  **True**

- and
- or
- not

## WHY?



## Operator Precedence

Logical operators are evaluated in this order:

1. `not`
2. `and`
3. `or`

You can override this order with parentheses

## Operator Precedence

Logical operators are evaluated in this order:

1. not
2. and
3. or

You can override this order with parentheses  
(True or False) and False  $\Rightarrow$



## Operator Precedence

Logical operators are evaluated in this order:

1. not
2. and
3. or

You can override this order with parentheses  
(True or False) and False  $\Rightarrow$  **False**

## Short-Circuit Evaluation

X: Any boolean value

True or X  $\Rightarrow$

## Short-Circuit Evaluation

X: Any boolean value

True or X  $\Rightarrow$  **True**

## Short-Circuit Evaluation

X: Any boolean value

True or X  $\Rightarrow$  **True**

False and X  $\Rightarrow$

## Short-Circuit Evaluation

X: Any boolean value

True or X  $\Rightarrow$  **True**

False and X  $\Rightarrow$  **False**



## Short-Circuit Evaluation

X: Any boolean value

True or X  $\Rightarrow$  **True**

False and X  $\Rightarrow$  **False**

Python is smart enough to take advantage of this!

## Short-Circuit Evaluation

X: Any boolean value

True or X  $\Rightarrow$  **True**

False and X  $\Rightarrow$  **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
```

# Truthy & Falsy Values

```
# 'Falsy' values
bool(None) # => False
bool(False) # => False
bool(0) # => False
bool(0.0) # => False
bool('') # => False
# Empty data structures
bool([]) # => False
```





# Truthy & Falsy Values

```
# 'Falsy' values
bool(None) # => False
bool(False) # => False
bool(0) # => False
bool(0.0) # => False
bool('') # => False
# Empty data structures
bool([]) # => False
```

```
# Everything else is 'truthy'
bool(-100000) # => True
bool('False') # => True
bool(3.14) # => True
bool(int) # => True
# Nonempty data structures
bool([1, 'a', []]) # => True
bool([False]) # => True
```

# Comparison Operators

- <: Strictly less than

## Comparison Operators

- <: Strictly less than
- <=: Less than or equal

## Comparison Operators

- <: Strictly less than
- <=: Less than or equal
- >: Strictly greater than

## Comparison Operators

- <: Strictly less than
- <=: Less than or equal
- >: Strictly greater than
- >=: Greater than or equal

## Comparison Operators

- <: Strictly less than
- <=: Less than or equal
- >: Strictly greater than
- >=: Greater than or equal
- ==: Equal

## Comparison Operators

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

## 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
```



## Chained Comparisons

$1 < 2 < 3 \Rightarrow$

# Chained Comparisons

`1 < 2 < 3 ⇒ True`

## Chained Comparisons

$1 < 2 < 3 \Rightarrow \text{True}$

You can chain arbitrarily many comparison operations together.

## Chained Comparisons

$1 < 2 < 3 \Rightarrow \mathbf{True}$

You can chain arbitrarily many comparison operations together.

$v_i$ : variables/values,  $op_i$ : comparison operators

## Chained Comparisons

$1 < 2 < 3 \Rightarrow \text{True}$

You can chain arbitrarily many comparison operations together.

$v_i$ : variables/values,  $op_i$ : comparison operators

$v_1 \ op_1 \ v_2 \ op_2 \ v_3 \ \dots \ op_{n-1} \ v_n$  is equivalent to:

$v_1 \ op_1 \ v_2 \ \mathbf{and} \ v_2 \ op_2 \ v_3 \ \mathbf{and} \ \dots v_{n-1} \ op_{n-1} \ v_n$

## Chained Comparisons

`1 < 2 < 3`  $\Rightarrow$  **True**

You can chain arbitrarily many comparison operations together.

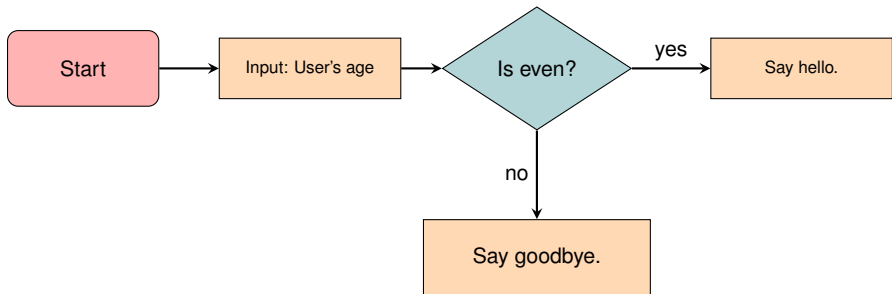
$v_i$ : variables/values,  $op_i$ : comparison operators

$v_1 \ op_1 \ v_2 \ op_2 \ v_3 \ \dots \ op_{n-1} \ v_n$  is equivalent to:

$v_1 \ op_1 \ v_2$  **and**  $v_2 \ op_2 \ v_3$  **and**  $\dots v_{n-1} \ op_{n-1} \ v_n$

```
3 > 2 == 1 < 5 > 4 # => False
3 > (2 == 1) < 5 > 4 # => True
3 > True > False # => True
3 > 5 < 1/0 # => False
3 < 5 < 1/0 # => ZeroDivisionError
```

# Branching



# 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?