1. Recap
○○○○

2. Basic Operators
○○○○○○○○○

3. Branching
○○

4. Objects
○○○○

5. Basic Functions
○○○○○

KOLT Python
Basic Operators, Intro to Branching & Simple Functions

Ceren Kocaoğullar

Tuesday 4th February, 2020

KOÇ UNIVERSITY
OFFICE OF LEARNING AND TEACHING

1. Recap
OOOO

2. Basic Operators
OOOOOOOOO

3. Branching
OO

4. Objects
OOOO

5. Basic Functions
OOOOO

Agenda

1. Recap

2. Basic Operators

3. Branching

4. Objects

5. Basic Functions

1. Recap
●○○○

2. Basic Operators
○○○○○○○○○

3. Branching
○○

4. Objects
○○○○

5. Basic Functions
○○○○○

Comments

```python
# 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**!

1. Recap
○●○○

2. Basic Operators
○○○○○○○○○

3. Branching
○○

4. Objects
○○○○

5. Basic Functions
○○○○○

Variables

Type	Explanation	Examples
`int`	represent **integers**	3, 4, 17, -10
`float`	represent **real numbers**	3.0, 1.11, -109.123123
`bool`	represent **boolean** 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?

1. Recap
OO●O

2. Basic Operators
OOOOOOOOO

3. Branching
OO

4. Objects
OOOO

5. Basic Functions
OOOOO

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')

1. Recap
OOO●

2. Basic Operators
OOOOOOOOO

3. Branching
OO

4. Objects
OOOO

5. Basic Functions
OOOOO

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`

- `+`: 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 # => 2.33333335
3.5 // 1.5 # => 2.0
3.5 % 1.5  # => 0.5
2 ** 10  # => 1024
```

1. Recap
0000

2. Basic Operators
0●0000000

3. Branching
00

4. Objects
0000

5. Basic Functions
00000

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 shorter representations for these updates with arithmetic operators.
`num += 1` is equivalent to `num = num + 1`
`result *= 2` is equivalent to `result = result * 2`

1. Recap
0000

2. Basic Operators
00●000000

3. Branching
00

4. Objects
0000

5. Basic Functions
00000

Assignment Operators

Operator	Usage	Equivalent
+=	`val += 3`	`val = val + 3`
-=	`val -= 3`	`val = val - 3`
*=	`val *= 3`	`val = val * 3`
/=	`val /= 3`	`val = val / 3`
%=	`val %= 3`	`val = val % 3`
**=	`val **= 3`	`val = val ** 3`
//=	`val //= 3`	`val = val // 3`

1. Recap
○○○○

2. Basic Operators
○○○●○○○○○

3. Branching
○○

4. Objects
○○○○

5. Basic Functions
○○○○○

`bool` **Operators**

How to represent logical operations in Python?

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**

`True or False and False` $\Rightarrow$ **True**

WHY?

1. Recap
OOOO

2. Basic Operators
OOOO●OOOO

3. Branching
OO

4. Objects
OOOO

5. Basic Functions
OOOOO

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 ⇒ **True**
False and X ⇒ **False**
Python is smart enough to take advantage of this!

```python
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

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

```python
# 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
```

1. Recap
OOOO

2. Basic Operators
OOOOOOO●O

3. Branching
OO

4. Objects
OOOO

5. Basic Functions
OOOOO

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   # => False
3 > 'a'   # => TypeError
3 == 'a'   # => False
```

1. Recap
0000

2. Basic Operators
00000000●

3. Branching
00

4. Objects
0000

5. Basic Functions
00000

Chained Comparisons

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

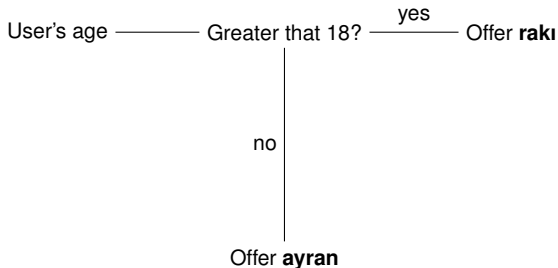You can chain arbitrarily many comparison operations together.

v_i: variables/values, op_i: comparison operators

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

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

```
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
```

1. Recap
○○○○

2. Basic Operators
○○○○○○○○○

3. Branching
●○

4. Objects
○○○○

5. Basic Functions
○○○○○

Branching

User's age —————— Greater that 18? ————— **yes** — Offer **rakı**

no

Offer **ayran**

Branching

```python
if <condition>:
    <expression>
    <expression>
    ...
```

```python
if <condition>:
    <expression>
    <expression>
    ...
else:
    <expression>
    <expression>
    ...
```

```python
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?

1. Recap
OOOO

2. Basic Operators
OOOOOOOOO

3. Branching
OO

4. Objects
●OOO

5. Basic Functions
OOOOO

Python Data Model

How did we represent data in Python?

Variables!

How do they work?
Do they store the data themselves?

1. Recap
oooo

2. Basic Operators
ooooooooo

3. Branching
oo

4. Objects
o●oo

5. Basic Functions
ooooo

Objects

Everything is an object in Python.

```
a = 5
a = 10
a += 3
print(a)
```
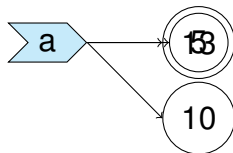

$\hookrightarrow$ Values at the right side of our label analogy are objects!
Even though variables **do not** have types, each object has
a **fixed** type.

1. Recap
0000

2. Basic Operators
000000000

3. Branching
00

4. Objects
00●0

5. Basic Functions
00000

Objects - Identity

Each object has an identity, this value can be obtained by using **id()** function.

== operator compares values
is operator compares identities

Objects - Identity

Is this glass half full or half empty?


```
# What fraction of this glass is water?
pessimist = 0.5
optimist = 0.5
pessimist == optimist      # => True
pessimist is optimist      # => False
```

Functions

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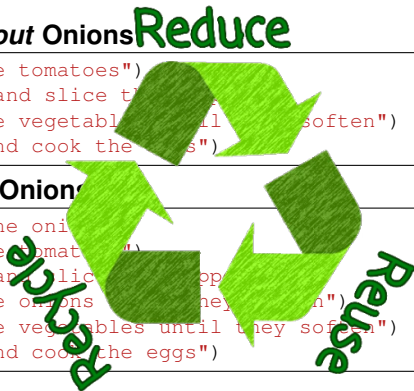
Functions

Menemen *without* Onions

```python
print("Chop the tomatoes")
print("Deseed and slice t
print("Cook the vegetabl        soften")
print("Crack and cook the      s")
```

Menemen *with* Onion

```python
print("Slice the oni
print("Chop the      mat
print("Deseed an      lic           op
print("Cook the   o  ey      n")
print("Cook the ve  tables until they sof      ")
print("Crack and co     the eggs")
```


Reduce

Recycle

Reuse

Defining Functions

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

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

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

Functions

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

Menemen *without* Onions

```
prepare_base_vegetables()
cook()
```

Menemen *with* Onions

```
print("Slice the onions")
prepare_base_vegetables()
print("Cook the onions until they soften")
cook()
```

Defining a function = **writing down the recipe**
Calling a function = **executing the recipe**

Functions

You **can** call a function inside another function.

```python
def menemen_with_onions():
    prepare_base_vegetables()
    cook()
```

```python
def menemen_without_onions():
    print("Slice the onions")
    prepare_base_vegetables()
    print("Cook the onions until they soften")
    cook()
```