Basic python

<table>
<tr><th colspan="2">Syntax</th></tr>
<tr><td rowspan="2">Indentation</td><td>- Code blocks are commonly started with:
- Indentation usually 4 spaces</td></tr>
<tr><td><pre>y = 0
for i in range(0, 3):
    y += i
    print(y)</pre></td></tr>
<tr><td rowspan="2">Comments</td><td>In-line
- #</td></tr>
<tr><td>Multi-line
- Not supported
- “””
- “”</td></tr>
<tr><td rowspan="2">Line Continuation</td><td>- Code too long</td></tr>
<tr><td>- Use line wrap:
- \</td></tr>
<tr><td rowspan="2">Multiple statements</td><td>- ;</td></tr>
<tr><td><pre>a = [1, 2, 3]; b = "my string"

is the same as
a = [1, 2, 3]
b = "my string"</pre></td></tr>
<tr><th colspan="2">Semantics</th></tr>
<tr><td rowspan="2">Variables
→ point to some object in the memory</td><td>- No need to declare variable type
- Variable’s type may change
- Type is linked to the object not to the variable</td></tr>
<tr><td>→ Python is a “dynamically typed language”
→ <u>not</u> type-free language</td></tr>
<tr><td>Arithmetic operators</td><td>- Addition: a + b
- Subtraction: a – b
- Multiplication: a * b
- Exponentiation: a ** b
- Division: a / b
- Floor division (quotient w/o fractional part): a // b
- Modulus: a % b
- Negation: -a
- Matrix product: a @ b</td></tr>
</table>

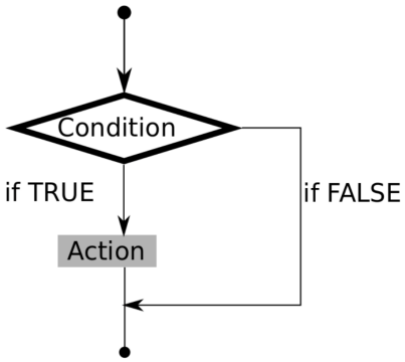
<table>
<tr><td>Comparison operators</td><td>
<table>
<tr><th>Code example</th><th>Description</th></tr>
<tr><td><code>a == b</code></td><td>a equal b</td></tr>
<tr><td><code>a != b</code></td><td>a not equal b</td></tr>
<tr><td><code>a < b</code></td><td>a less than b</td></tr>
<tr><td><code>a <= b</code></td><td>a less or equal b</td></tr>
<tr><td><code>a > b</code></td><td>a greater than b</td></tr>
<tr><td><code>a >= b</code></td><td>a greater or equal b</td></tr>
</table>
</td></tr>
<tr><td>Assignment operators</td><td>
<table>
<tr><th>Code example</th><th>Description</th></tr>
<tr><td><code>a += 5</code></td><td>Add And</td></tr>
<tr><td><code>a -= 5</code></td><td>Subtract And</td></tr>
<tr><td><code>a *= 5</code></td><td>Multiply And</td></tr>
<tr><td><code>a **= 5</code></td><td>Exponent And</td></tr>
<tr><td><code>a /= 5</code></td><td>Division And</td></tr>
<tr><td><code>a %= 5</code></td><td>Modulus And</td></tr>
<tr><td><code>a //= 5</code></td><td>Floor division And</td></tr>
</table>
</td></tr>
<tr><td>Logical operators</td><td>- AND
- NOR
- NOT
- <s>XOR</s></td></tr>
<tr><td rowspan="3">Membership operators</td><td>- Find values in list, tuples, or strings</td></tr>
<tr><td>- Return True or False</td></tr>
<tr><td>- IN
- NOT IN</td></tr>
<tr><td rowspan="2">Identity operators</td><td>- Compare memory location of objects</td></tr>
<tr><td>- IS
- IS NOT</td></tr>
<tr><td colspan="2">Data Types</td></tr>
<tr><td></td><td>- Bool
- String
- Number:
  ○ Integer (signed integers)
  ○ Float (floating point numbers in double precision)
  ○ Complex (complex numbers with real and imaginary part)
- List
- Tuple
- Set
- Dictionary</td></tr>
<tr><td>String</td><td>- Add
- Slice
- Repeat</td></tr>
</table>

	In [1]: `my_string = "Say something "` `print(my_string[0:3])  # prints the first three letters` `print(my_string[4:])  # prints the fifth to the last letter` `print(my_string[-2])  # prints the second last letter` `print(my_string + "- right now")  # appends "- right now"` `print(my_string * 2)  # prints the string twice` Say something g Say something - right now Say something Say something
Numbers	
Integers	- Division is automatically converted into floats
Floats	- With decimal point: a = 3.14 - With exponential notation: a = 1e-3 (1*10^(-3)) - Never use exact equality tests for floats → precision of Floats is limited
Lists	- [ ] - Entries and shape (length) of lists may change → mutable - ordered
	List comprehension - use to create lists with loops - [*expr for variable* in *iterable*] - [*expr for variable* in *iterable* if *condition*] In [2]: `# Create a list from 0 to 9, where each entry is the negative of the index` `[-i for i in range(10)]` Out[2]: `[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]` In [4]: `# Here with a condition` `[-i for i in range(10) if i % 2 == 0]` Out[4]: `[0, -2, -4, -6, -8]`
Tuple	- ( ) - Ordered - Immutable
Set	- Collections of unique values - Unordered - { } - Union, intersection, …
Dictionaries	- Unordered - Mutable - Key and value file ○ A = {“key_one”: value_one, “key_two”: value_two}

```
In [43]: a = {"temp": 300, "pressure": 15.78}
         a["pressure"]
Out[43]: 15.78

In [44]: # Read the key names
         a.keys()
Out[44]: dict_keys(['temp', 'pressure'])

In [45]: # Add a new field
         a["speed"] = 125

In [47]: a.keys()
Out[47]: dict_keys(['temp', 'pressure', 'speed'])

In [48]: # Print all values
         a.values()
Out[48]: dict_values([300, 15.78, 125])
```

Conditions

Control flow



- if
- elif (Python's version of else if)
- else

```
# Python treats zero as False
condition = 0

if condition:
    print("Catched")
else:
    print("Not catched")
```

```
Not catched
```

```
# Bools work of course perfectly

condition = True

if condition:
    print("Catched")
else:
    print("Not catched")
```

```
Catched
```

```
# Any arithmetic or comparision operator will work as well

signal = 5.5

if signal > 2.5:
    print("Signal overshoot")
```

```
Signal overshoot
```

Loops

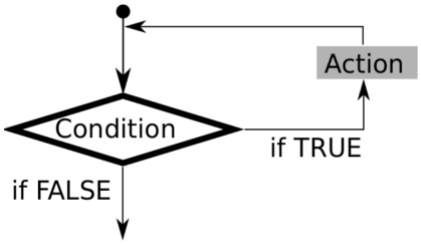
- basic loops

<table>
<tr><td></td><td>

while
for

- finer control inside loops provided by

break
continue
pass statements

</td></tr>
<tr><td></td><td> 
</td></tr>
<tr><td>Iterators</td><td>
- in range(start, stop)
- in
- in enumerate

enumerates many objects like lists and returns index and item

</td></tr>
<tr><td>For loop</td><td>
- repeats a statement over a sequence
</td></tr>
<tr><td></td><td>

```
for i in [0, 2, 3]:
    print(i)
```

```
0
2
3
```

```
for i in range(2, 5):
    print(i)
```

```
2
3
4
```
</td></tr>
<tr><td></td><td>

```
# You can also loop over dictionaries

my_dict = {"power": 3.5, "speed": 120.3, "temperature": 23}

for field in my_dict.keys():
    print(field, "is adjusted to", my_dict[field])
```

```
power is adjusted to 3.5
temperature is adjusted to 23
speed is adjusted to 120.3
```
</td></tr>
<tr><td></td><td>
- Floor loop over two lists

zip

</td></tr>
<tr><td></td><td>

```
list_one = [0, 3, 5]
list_two = [8, 7, -3]

for i, j in zip(list_one, list_two):
    print(i * j)
```

```
0
21
-15
```
</td></tr>
<tr><td>While loop</td><td>
- repeats a statemnet as long as condition is true
- used if you do not know how long the sequence should be repeated
- condition is checked before code execution
- less used than for loops
</td></tr>
</table>

```
i = 0

while i < 4:
    i += 1  # do not forget to increment
    print(i)
```

```
1
2
3
4
```

Functions

Clean code	- A function does ONE thing
Basic syntax	def function_name (argument): """Doc string.""" code return something
Arguments	- Required arguments - Keyword arguments - Default arguments - Arguments of variable length
	Default arguments - Functions can either be called with or without the default
Variable length arguments	- *args - **kwargs → collect all arguments and keyword arguments
	- Arguments prefixed with * are converted to a sequence - Arguments prefixed with ** are converted to a dictionary
Multiple returns	- Not very clean coding
Syntax	- return var1, var2 → return becomes a tuple

```
def mult_return(x, y):
    return x**2, y**2


mult_return(3, 5)
```

```
# First return
mult_return(3, 5)[0]
```

Anonymous functions	- lambda functions for one linear function - Just return one value - Cannot operate multiple expressions - Cannot access global variables
Syntax	- lambda arg, arg: expression

```
my_sum = lambda x, y: x + y
diff = lambda x, y: x - y

def my_print(x, y, func):
    print(func(x, y))
```

```
my_print(2, 2, func=my_sum)
```

```
4
```

- should be avoided

	`` my_sum = lambda x, y: x + y # can be rewritten as def my_sum(x, y): return x + y ```
Type hints	- Tell other developers/code checkers what kind of variable type a function wants to get - Good for large projects with many different developers - Optional
	`` def greeting(name: str) -> str: print('Hello ', name) ``` Here, the `(name: str) -> str:` tells that this function wants to see strings. This does not mean that cannot pass "wrong" data type. ``` greeting(0) ``` ``` Hello 0 ``` ``` greeting("Peter") ``` ``` Hello Peter ```
Classes	
Object oriented programming	- Objects: something that has some properties and cam do some things
	`` class Engine: """A class that defines an engine with some methods.""" conversion = 3.6 def __init__(self, hp, consumption): self.power = hp self.cons = consumption def mps_in_kph(self, mps): """Here a method which converts meter per second in kilometer per hour.""" return self.conversion * mps def get_power(self): """This method prints the power of the engine.""" print(self.power) def consumption(self, distance): return self.cons * distance ```
Syntax	- Defined with class
Attributes	- Most public (read- or writeable - Protect attributes: - Protected attributes: variable name prefixed with <u>one</u> underscor (_varname) - Private attributes: prefixed with <u>two</u> underscores (__varname)
	- Print doc string of object with `small_eng.__doc__`

<table>
<tr><td></td><td>
- Access attributes of object
<code>getattr(small_eng, 'conversion')</code>
Or
<code>small_eng.conversion</code>
- Use method of class
<code>small_eng.get_power()</code>
Or
<code>small_eng.consumption(distance=5.8)</code>
- Change property
<code>large_eng.cons = 11.9</code>
Changes bevahior:
<pre>large_eng.consumption(distance=5.8)</pre>
<pre>69.02</pre>
</td></tr>
<tr><td>Inheritance</td><td>
<pre>class Parent:
    income = "large"

class Child(Parent):
    def education_level(self):
        if Parent.income == "large":
            print("High education level")
        else:
            print("Minor education level")</pre>
<pre>peter = Child()
peter.education_level()</pre>
<pre>High education level</pre>
- Child knows attribute of the Parent class (class Child(Parent))
</td></tr>
<tr><td colspan="2">Modules</td></tr>
<tr><td>Explicit import</td><td>
- import
- Access methods: .
</td></tr>
<tr><td>Import with alias</td><td>
- import statistics as stat
</td></tr>
<tr><td>Import specific module content</td><td>
- from statistics import mean as my_mean
</td></tr>
<tr><td>Implicit import</td><td>
- with aterisk *
- could override some older imports or self defined functions
- from module import *
</td></tr>
<tr><td>Self-written modules</td><td>
- __init__.py
</td></tr>
<tr><td></td><td>
<pre>scritpt.py # a script which imports the module
src # a folder
  | __init__.py # empty file
  | my_module.py # your module which contains for
instance some functions</pre>
</td></tr>
</table>