TECHNISCHE
UNIVERSITÄT
DRESDEN

PYTHON BASICS: SYNTAX AND DATA-TYPES

Material from Ingo Keller, edited by Christoph Pohl and Carsten Knoll

Dresden, 2017-10-07

Numerical Data Types

- Integer

```
>>> type(1)
<type 'int'>
```

- floating point number

```
>>> type(1.0)
<type 'float'>
```

- complex number

```
>>> type(1 + 2j)
<type 'complex'>
```

- Operations

Addition	`+`
Subtraction	`-`
Division	`/`
Integer division	`//`
Multiplication	`*`
Taking powers	`**`
Modulo	`%`

- Built-in functions
  - `round`, `pow`, etc.
  - see `dir(__builtins__)`
- Module `math`
  - see `help(math)`

NoneType and boolean values

- `None`
  - universal value for “undefined”

```
>>> type(None)
<type 'NoneType'>
```

- Boolean values
  - `True` and `False`

```
>>> type(True)
<type 'bool'>
```

Data Type	False-Value
`NoneType`	`None`
`int`	`0`
`float`	`0.0`
`complex`	`0 + 0j`
`str`	`""`
`list`	`[]`
`tuple`	`()`
`dict`	`{}`
`set`	`set()`

Operations

Operation	Shortcut
`x = x + y`	`x += y`
`x = x - y`	`x -= y`
`x = x * y`	`x *= y`
`x = x / y`	`x /= y`
`x = x % y`	`x %= y`
`x = x ** y`	`x **= y`
`x = x // y`	`x //= y`

Comparison operations
`x == y`
`x != y`
`x < y`
`x <= y`
`x > y`
`x >= y`

Strings (objects of type str)

```
str1 = "abc"
str2 = 'xyzabcefghi'
str3 = """
      multi
      line
      string
      """
```

```
>>> str2[0] # 0 is first index
'x'
>>> str2[1:4]
'yza'
>>> str2[-3:]
'ghi'
```

Escape Sequence	Meaning
\n	newline
\r	carriage return
\"	escaping "
\'	escaping '
\\	escaping \\

String Formating

- General Syntax

```
"value of x={} and y={}".format(x, y)
```

- Examples

```
>>> a = 'H'
>>> b = 'ello World'
>>> "{}{}{} {0}".format(a, b, 5)
>>> "{}{}{} {0}".format(a, b, 5)
'Hello World'
```

- Extension (see also: reference)

```
>>> "a={:06.2f} and b={:05.2f}".format(3.007, 42.1)
'a=003.01 and b=42.10'
```

- important methods of class `str`:

```
index, replace, split, join,
format, startswith, endswith, ...
```

Lists

- Syntax
  `[value_1, ..., value_n]`
- Can contain values of any type
- Can be changed
- Can be sorted
- Important methods
  `append, count, index, insert, pop, remove, reverse, sort`

⚠ `sort` and `reverse` work „in place“
(return-value: `None`)

- Examples

```
>>> m = [7, 8, 9]
>>> n = ['a', 'z', 1, False]
>>> m.append('x')
>>> m[0]
7
>>> m[-1]
'x'

>>> m[:] # start to end
[7, 8, 9, 'x']
>>> m.pop(0)
7
>>> m.reverse()
>>> print(m)
['x',9,8]
```

Tuple

- Syntax
  `(value_1, ..., value_n)`
- Can **not** be changed
- → Access much faster that to `list`
- Can contain elements of any type
- important methods
  `index`

- Examples

```
>>> t = (7,8,9)
>>> t[0]
7
>>> t[-1]
9
>>> t[:] # start to end
(7,8,9)
>>> z = ('a', 'z', 1, False)
>>> t.index(8)
1
>>> z.index('a')
0
```

Sequential data types

`str`, `tuple`, `list`, (`numpy.array`)

Operation	Meaning
`s in x`	tests, whether s is element of x
`s not in x`	tests, whether s is not element of x
`x + y`	concatenation of x and y
`x * n`	concatenation, such that n copies of x exist
`x[n]`	return the n-th element of x
`x[n:m]`	return the subs-sequence from index n til m (excluding m)
`x[n:m:k]`	same with step-size k
`len(x)`	number of elements
`min(x)`	minimum
`max(x)`	maximum

Dictionaries (Associative Arrays)

- Key-value-pairs
  - Keys must be immutable objects
  - Each key can occur only once
- Syntax

```
{ Key_1: Value_1,
  Key_2: Value_2,
  ... }
```

- Access via
  - `d.get(key, default)`
    or
  - `d[key]`
- Important methods
  - `keys, values, items`

Examples

```
>>> d = { "Germany": "Berlin", "Peru": "Lima"}

>>> type(d)
<type 'dict'>

>>> e = {1: "a", 2: "b", 400: "c", 1.3: d}
>>> e[1]
'a'

>>> d.get("Germany")
'Berlin'

# no entry -> None (no output)
>>> d.get("Bavarya") # -> None

# with default value
>>> d.get("Bavarya", "unknown capital")
'unknown capital'

>>> d["Bavaria"]
KeyError: 'Bavaria'
```

Sets

- Syntax
  `set([element_1, ..., element_n])`
- Every element is contained only once
- Has no specified order
- Can be changed (`frozenset` is immutable)
- Important methods:
  `add, remove, union, difference, issubset, issuperset`

Examples

```
>>> engineers = set(['Jane', 'John'
... 'Jack', 'Janice'])
>>> programmers = set(['Jack', 'Sam',
... 'Susan', 'Janice'])
>>> managers = set(['Jane', 'Jack',
... 'Susan', 'Zack'])
>>> s1 = engineers.union(programmers)
>>> s2 = engineers.intersection(managers)
>>> s3 = managers.difference(engineers)
>>> engineers.add('Marvin')
>>> print(engineers)
set(['Jane', 'Marvin',
'Janice', 'John', 'Jack'])
```

Data Types - Final Remarks

- Everything in Python is an object (even functions, classes, modules)

→ Everything has a type: `type(object)`

- Type checking (→ `True` or `False`):
  - Exact matching: `type("abc") == type("xyz")`
  - Better: respecting inheritance `isinstance(x, str)`
  - Allow multiple types: `isinstance(x, (int, float, complex))`

- Useful construction: `assert isinstance(x, int) and x > 0`

Distinction of Cases: `if`, `elif`, `else`

- Syntax

```
he indention
if <condition1>:
    ...
elif <condition2>:
    ...
else:
    ...
```

- Examples

```
>>> x = 1
>>> if x == 1:
...     print("x is 1")
...
x is 1
>>> x = 4
>>> if x == 1:
...     print("x is 1")
... elif x == 3:
...     print("x is 3")
... else:
...
print("x is neither 1 nor 3")
x is neither 1 nor 3
```

Iterate over a Sequence: `for`-loop

- Syntax:

```
for <variable> in <sequence>:
    ...
```

- easily construct sequences:
- `range`-function → iterator

```
range(stop)
range(start, stop)
range(start, stop, step)

>>> list(range(4))
[0, 1, 2, 3]

>>> list(range(1, 10, 2))
[1, 3, 5, 7, 9]

# conversion to list only for printing
```

- Examples:

```
>>> seq = ['a', 'b', 42]
>>> count = 0
>>> for elt in seq:
...     print(elt*2)
aa
bb
84

>>> for i in range(3):
...     print(2**i)
1
2
4
```

Loop `while` condition is true

- Syntax

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

- `break`
  terminates the loop

```
while <condition1>:
    if <condition2>:
        break
```

- `continue`
  immediately starts next cycle

```
while <condition1>:
    if <condition2>:
        continue
```

- Examples

```
>>> x = 4
>>> while x > 1:
...     print(x)
...     x -= 1
... print("finished")
4
3
2
finished
```

Functions

- Syntax

```
def func_name(Param_1, ..., Param_n):
    ...
    return <result>
```

- No explicit return-value → `None`
- Empty function with keyword `pass`:

```
def empty():
    pass
```

- default values for optional parameters

```
def test(x=23):
    print(param)
```

- Arbitrary number of arguments

```
def func(*args, **kwargs):
    print(type(args)) # -> tuple
    print(type(kwargs)) # -> dict
```

- Examples

```
>>> def print_sum(a, b):
...     print(a + b)
>>> print_sum(1, 2)
3
>>> def print_prod(a, b, c=0):
...     print(a*b + c)
>>> print_prod(2, 4)
8

# better readable
>>> print_prod(a=2, b=4)
8
>>> print_prod(2, 4, 1)
9
>>> print_prod(c=2, a=4, b=1)
6
```

Local Variables (Scopes)

Listing: local-variables.py

```python
def square(z):
    x = z**2 # x: local variable
    print(x)
    return x

x, a = 5, 3 # "unpacking" a tuple

square(a) # -> 9
square(x) # -> 25
print(x) # -> 5 (not changed)

def square2(z):
    print(x) # here: x is taken from global scope
    return z**2

def square3(z):
    print(x) # Error (local variable not yet known)
    x = z**2 # x is local variable due to write access
    return x
```

General Syntax

- Semantic blocks are defined by indention level (in place of, e.g., `{ ... }`)
  - defacto-standard: 4 spaces per level (do not use TABs)
  - every good text editor can be configured adequately (spyder: `TAB` indention, `SHIFT+TAB` dedetion of highlighted lines)
- Comments and docstrings:

```
# single line comments begin with a hash

def my_function(x, y):
s is a docstring.
 span multiple lines


ssigned multi-line strings can
sed as multi-line comment
```

- Recommended maximum line length 80 (or 100) characters (readability)
- If you need more:
  - Check possibility to split up into two commands (readability)
  - Within braces newlines are ignored
  - Backslash (\) allows line continuation in expression

Keywords (Reserved words)

`False`
`None`
`True`
`and`
`as`
`assert`
`break`
`class`
`continue`
`def`
`del`
`elif`
`else`
`except`
`finally`
`for`
`from`
`global`
`if`
`import`
`in`
`is`
`lambda`
`nonlocal`
`not`
`or`
`pass`
`raise`
`return`
`try`
`while`
`with`
`yield`

They cannot be used as variable name or similar.

File Access

Listing: file-access.py

```python
# write in text mode
content_lines = ['some\n', 'more', 'content']
with open('text.txt', 'w') as myfile:
    myfile.write('Hello World.')
    myfile.writelines(content_lines)
    # myfile.close() is called automatically
    # when leaving this block

# read in text mode
with open('text.txt', 'r') as myfile:
    header = myfile.read(10) # first 10 byte
    lines = myfile.readlines() # list of lines
    # (starting from file cursor)
```

File Access

Listing: file-access.py

```python
# write in text mode
content_lines = ['some\n', 'more', 'content']
with open('text.txt', 'w') as myfile:
    myfile.write('Hello World.')
    myfile.writelines(content_lines)
    # myfile.close() is called automatically
    # when leaving this block

# read in text mode
with open('text.txt', 'r') as myfile:
    header = myfile.read(10) # first 10 byte
    lines = myfile.readlines() # list of lines
    # (starting from file cursor)
```

Read/write binary data: use `'rb'` and `'wb'`
Appending text or binary data: use `'a'` or `'ab'`

Some “specialities” of Python

- Idexing starts with 0
- Unpacking of sequential data types:

```
>>> x, y, z = range(3)
>>> y
1

>>> mapping = [('green', 560), ('red', 700)]
>>> for color, wavelength in mapping:
...     pass
...     # do stuff
```

- ∃ extensive standard library („batteries included“)
  - http://docs.python.org/3/library/

  → “Don’t reinvent the wheel!”

  - Important modules: `pickle, sys, os, itertools, unittest, ...`

Links

- Official tutorial: `http://docs.python.org/3/tutorial/`
- Interactive tutorial: `http://www.learnpython.org/`

- Extensive well structured course: `http://www.diveintopython3.net/`