Technische Universität Braunschweig

IAS | INSTITUTE FOR APPLICATION SECURITY

Introduction to Python

WiSe 25/26

Manuel Karl & Jan Drescher, 2025-10-23

Python Philosophy

- **Philosophy** (PEP 20: The Zen of Python)
  - Beautiful is better than ugly
  - Simply is better than complex
  - Complex is better than complicated
  - Readability counts

Types and Variables

- Python is
  - dynamically typed and uses duck-typing (types checked at runtime)
  - strongly typed (only well-defined operations)
- Values/Objects are typed but variables are not

```
s = "I'm a string" # string
n = len(s) # number
```

Lists and Tuples

- **Lists** can contain items of arbitrary type

```
l1 = ["s", "e", "c"]
l2 = ["tu-braunschweig", 20, 1.9]
l3 = l1 + l2
l3[3] = "tu-bs"
l3[-1] = 19
```

- **Tuples** are similar to lists but are immutable

```
t1 = ("tu-bs", 20, 1.9)
t1[2] = 19
TypeError: 'tuple' object does not support item assignment
```

Dictionaries

- Associated arrays
  - Sequences (lists, tuples) are indexed by numbers
  - Dictionaries can be indexed by any (immutable) type

```
d = { 1: "sec", "tu-bs": "de", 20: 1.9 }
```

- Accessing the data
  - Individual items: `d[1]`, `d["tu-bs"]`, `d[20]`
  - All keys `d.keys()`: `[1, "tu-bs", 20]`
  - All values `d.values()`: `["sec", "de", 1.9]`
  - All items `d.items()`: `[(1, "sec"), ... ]`

if statements

- No brackets necessary
  - Conditions are limited by keyword and : `if a == b:`
- No braces necessary
  - Statement blocks are marked by indentation

```
if a == b:
    # your code here
else:
    # more code here
```

- Ternary if `c = ('equal' if a == b else 'unequal')`

Conditions

- Operators known from other languages
  - Comparison `a>b`, `a>=b`, `a<b` & `a<=b`
  - Equality `a==b` & `a!=b`
  - Logical operators `not`, `or` & `and`
- but there is more
  - Freaky comparison `a < b <= c`
  - Membership operator `in` & `not in`
  - Identity operator `is` & `is not`

match statement

```
day = 4
match day:
  case 6:
    print("Today is Saturday")
  case 7:
    print("Today is Sunday")
  case _:
    print("Looking forward to the Weekend") # default case
```

Loops

- For loop over iterable/sequence or generator expression:

```
for i in [1,2,3]:
    # your code here
```

```
for i in range(1,4):
    # your code here
```

- While loop

```
while a == 0:
    # your code here
```

- early exit: `break`
- jump to loop header: `continue`

Loops with else clauses

- For Else

```
for i in [1,2,3]:
    # your code here
else:
    # Not executed after a break
    print("Last element: {}".format(i))
```

- While Else

```
while len(stack) > 0:
    # your code here
else:
    print("All processed")
```

Functions

- Declaration using the `def` keyword

```
def my_function(a, b, c=None):
    # your code here
    return 0.1
```

```
my_function(1, 2)
my_function(b=2, a=1, c=3)
my_function('1', '2', '3')
```

- Lambda functions

```
my_function = lambda a, b, c: # your code here
```

Classes

- Fields, member functions, inheritance,

```
class Foo(Bar1, Bar2):
    """Class description"""

    def __init__(self, a, b, c):
        """Constructor"""
        self.a = a # public
        self._b = b # protected
        self.__c = c # private

    def my_function(x, y):
        return x * y + self.__c
```

Exceptions

- Its better to ask forgiveness than permission

```
try:
    raise TypeError("Cannot process strings")
except:
    # your exception handling code here
```

- Explicity state the exception to catch

```
try:
    z = 1/0
except ZeroDivisionError as e:
    print("Divide-by-zero: {}".format(e))
```

Advanced Exception Handling

```
try:
    # your risky code here
except (ValueError, TypeError):
    rollback()
    raise TypeError("Illegal input data")
except:
    rollback()
    raise
else:
    commit() # executed if everything went well
finally:
    print("End of operation") # always executed
```

Importing Modules

- Python code is organized in modules
  - Each `*.py` file is one module

```
import foo
f = foo.Foo(0, 1, 2)
x = f.my_function(4, 2)
```

- Importing specific functions/classes only

```
from foo import Foo
f = Foo(0, 1, 2)
x = f.my_function(4, 2)
```

Python Packages

- Modules are organized as packages
  - Packages are directories containing modules
  - Must contain a `__init__.py` module
- Python offers a huge standard library
  - Documentation at https://docs.python.org/3/library/

```
import os.path as path
filename = path.basename("/usr/bin/python")
pypath = path.join("", "usr", "bin", filename)
```

def main()

- Use a dedicated main function

```
def main():
    # your code here

if __name__ == "__main__":
    main()
```

- Why is this a good thing to do?
  - Importing the module wont execute code -but it is still possible:
    `import module; module.main()`

Comments and Documentation

- Simple comments: `# Just use the hashtag`
- Docstring convetions (PEP 257)
  - Triple double quotes: `"""Im a docstring"""`
  - One-line and multi-line documentation possible

```
def point(x=0.0, y=0.0):
    """Form a 2D point.
    Keyword arguments:
    x -- x-coordinate (default: 0.0)
    y -- y-coordinate (default: 0.0)
    """
```

Formatting

- Classes use **CapWords** style: `class MyClass:`
- Functions and variables use **snake_case** style: `def my_beautiful_function():`
- Tip: Use a linter like flake8 or pylint, or a formatter like black

Typehints

- You can add typehints to functions and variables
  - https://peps.python.org/pep-0484/
- Typehints are optional
- They allow the linter to point out type errors
- For complex types (e.g., lists, unions) use the typing library

```python
def greeting(name: str) -> str:
    s: str = "Hello"
    s += name
    return s
```

Syntactic Sugar

- List/Dictionary comprehensions
  - `[x**2 for x in range(5)]: [0, 1, 4, 9, 16]`
  - `{x: x**2 for x in range(5)}: {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}`
- Format strings

```
age = 25
name = "John"
print(f"{name} is {age} years old")
>>> John is 25 years old
```