Introduction to Python

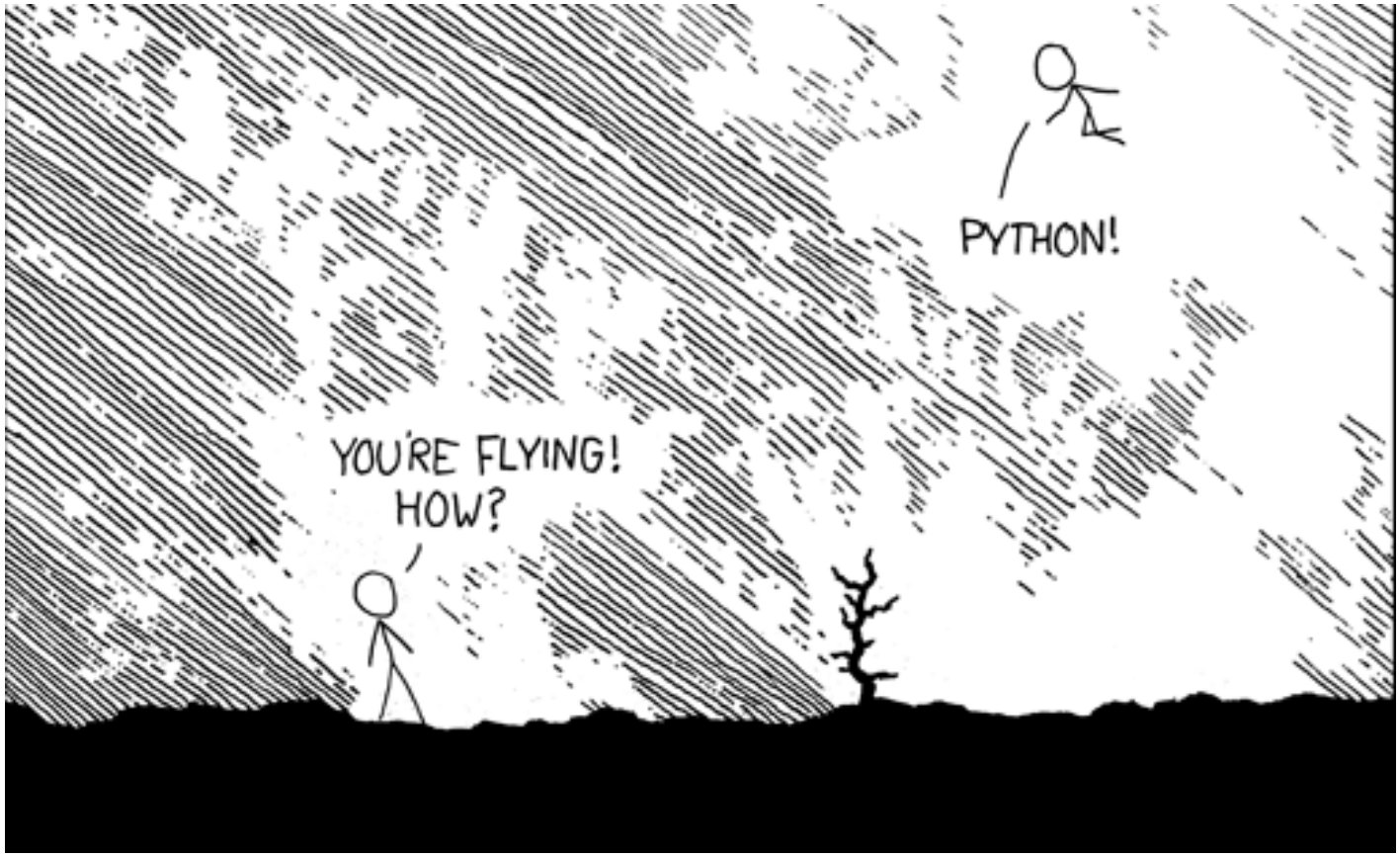
Petr Zemek

Senior Developer at Avast Software
Threat Labs (Viruslab)
petr.zemek@avast.com
https://petrzemek.net

Principles of Programming Languages, BUT FIT, 2019-03-04

Motto

"Python makes you fly."



https://xkcd.com/353/

Why Python? Whetting our Appetite

Feb 2019	Feb 2018	Change	Programming Language	Ratings	Change
1	1		Java	15.876%	+0.89%
2	2		C	12.424%	+0.57%
3	4	▲	Python	7.574%	+2.41%
4	3	▼	C++	7.444%	+1.72%
5	6	▲	Visual Basic .NET	7.095%	+3.02%
6	8	▲	JavaScript	2.848%	-0.32%
7	5	▼	C#	2.846%	-1.61%
8	7	▼	PHP	2.271%	-1.15%
9	11	▲	SQL	1.900%	-0.46%
10	20	⇈	Objective-C	1.447%	+0.32%

http://www.tiobe.com/tiobe-index/

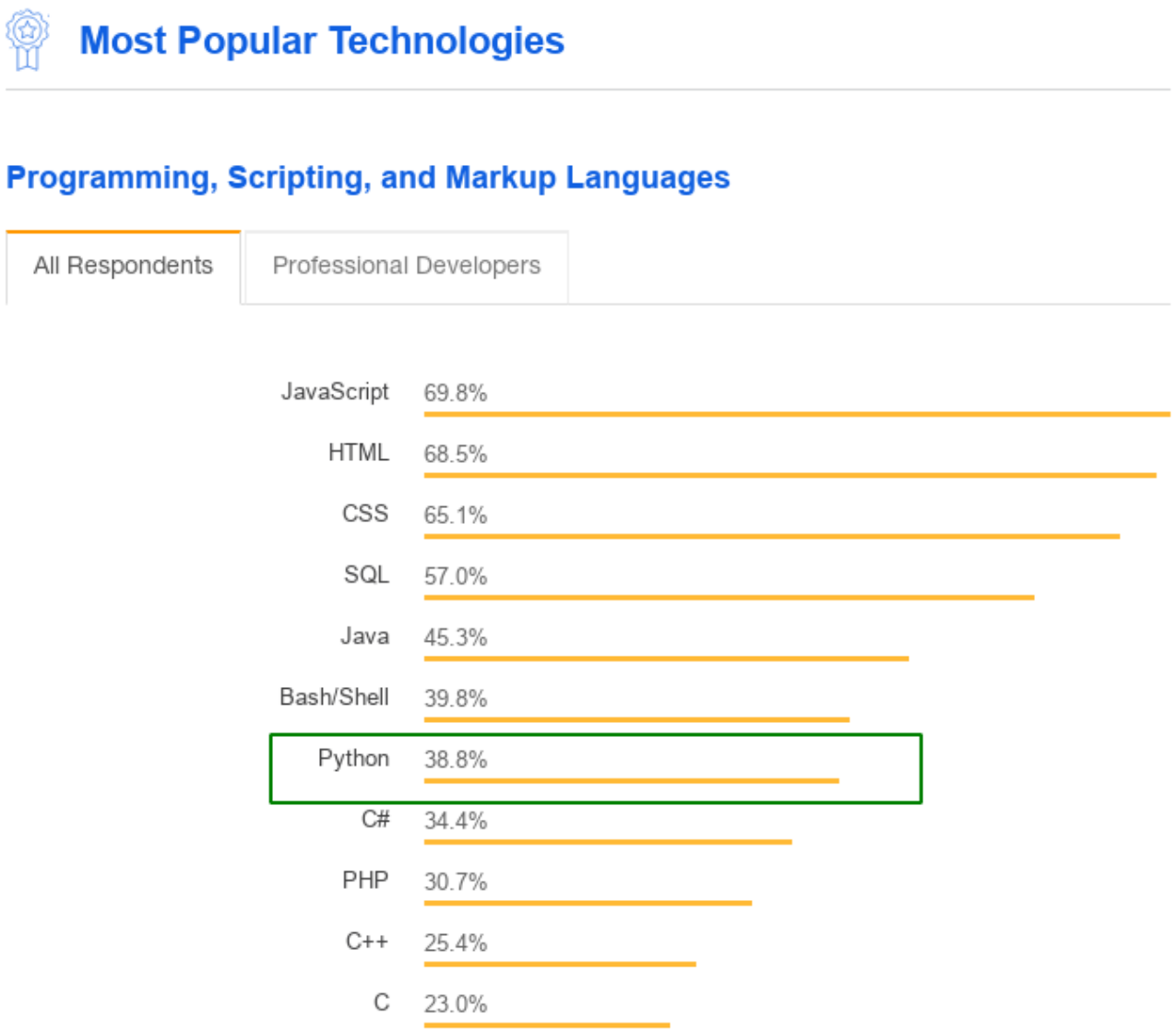
Why Python? Whetting our Appetite

Worldwide, Feb 2019 compared to a year ago:

Rank	Change	Language	Share	Trend
1	↑	Python	26.42 %	+5.2 %
2	↓	Java	21.2 %	-1.3 %
3	↑	Javascript	8.21 %	-0.3 %
4	↑	C#	7.57 %	-0.5 %
5	↓↓	PHP	7.34 %	-1.2 %
6		C/C++	6.23 %	-0.3 %
7		R	4.13 %	-0.1 %
8		Objective-C	3.04 %	-0.8 %
9		Swift	2.56 %	-0.6 %
10		Matlab	1.98 %	-0.4 %

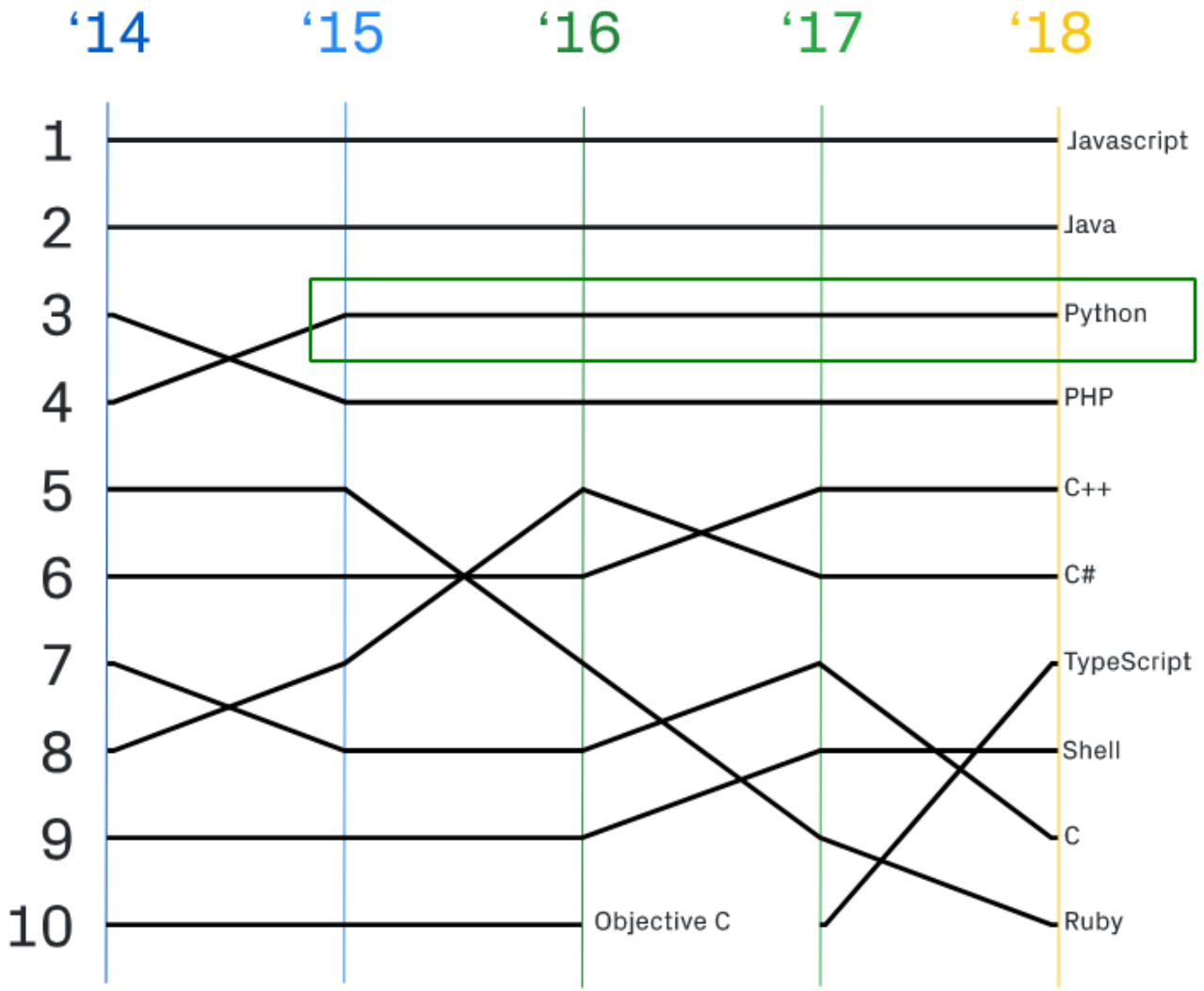
http://pypl.github.io/

Why Python? Whetting our Appetite



https://insights.stackoverflow.com/survey/2018

Why Python? Whetting our Appetite



https://octoverse.github.com/projects#languages

What is Python?

- widely used, general-purpose high-level programming language

What is Python?

- widely used, general-purpose high-level programming language
- design philosophy emphasizes code readability

What is Python?

- widely used, general-purpose high-level programming language
- design philosophy emphasizes code readability
- multiparadigm (procedural, object oriented)

What is Python?

- widely used, general-purpose high-level programming language
- design philosophy emphasizes code readability
- multiparadigm (procedural, object oriented)
- compiled to bytecode and interpreted in a virtual machine

What is Python?

- widely used, general-purpose high-level programming language
- design philosophy emphasizes code readability
- multiparadigm (procedural, object oriented)
- compiled to bytecode and interpreted in a virtual machine
- everything is an object

What is Python?

- widely used, general-purpose high-level programming language
- design philosophy emphasizes code readability
- multiparadigm (procedural, object oriented)
- compiled to bytecode and interpreted in a virtual machine
- everything is an object
- strongly, dynamically typed

What is Python?

- widely used, general-purpose high-level programming language
- design philosophy emphasizes code readability
- multiparadigm (procedural, object oriented)
- compiled to bytecode and interpreted in a virtual machine
- everything is an object
- strongly, dynamically typed
- duck typing

What is Python?

- widely used, general-purpose high-level programming language
- design philosophy emphasizes code readability
- multiparadigm (procedural, object oriented)
- compiled to bytecode and interpreted in a virtual machine
- everything is an object
- strongly, dynamically typed
- duck typing
- whitespace is significant

What is Python?

- widely used, general-purpose high-level programming language
- design philosophy emphasizes code readability
- multiparadigm (procedural, object oriented)
- compiled to bytecode and interpreted in a virtual machine
- everything is an object
- strongly, dynamically typed
- duck typing
- whitespace is significant
- portable (Windows, Linux, macOS)

What is Python?

- widely used, general-purpose high-level programming language
- design philosophy emphasizes code readability
- multiparadigm (procedural, object oriented)
- compiled to bytecode and interpreted in a virtual machine
- everything is an object
- strongly, dynamically typed
- duck typing
- whitespace is significant
- portable (Windows, Linux, macOS)
- many implementations (CPython, PyPy, Jython, IronPython)

What is Python?

- widely used, general-purpose high-level programming language
- design philosophy emphasizes code readability
- multiparadigm (procedural, object oriented)
- compiled to bytecode and interpreted in a virtual machine
- everything is an object
- strongly, dynamically typed
- duck typing
- whitespace is significant
- portable (Windows, Linux, macOS)
- many implementations (CPython, PyPy, Jython, IronPython)
- automatic memory management (garbage collector)

What is Python?

- widely used, general-purpose high-level programming language
- design philosophy emphasizes code readability
- multiparadigm (procedural, object oriented)
- compiled to bytecode and interpreted in a virtual machine
- everything is an object
- strongly, dynamically typed
- duck typing
- whitespace is significant
- portable (Windows, Linux, macOS)
- many implementations (CPython, PyPy, Jython, IronPython)
- automatic memory management (garbage collector)
- free (both as in "free speech" and "free beer")

A Glimpse at Python History

- invented in the beginning of 1990s by Guido van Rossum

A Glimpse at Python History

- invented in the beginning of 1990s by Guido van Rossum

A Glimpse at Python History

- invented in the beginning of 1990s by Guido van Rossum

- its name stems from "Monty Python's Flying Circus"

A Glimpse at Python History

- invented in the beginning of 1990s by Guido van Rossum

- its name stems from "Monty Python's Flying Circus"
- version history:
  - Python 1.0 (January 1994)

A Glimpse at Python History

- invented in the beginning of 1990s by Guido van Rossum

- its name stems from "Monty Python's Flying Circus"
- version history:
  - Python 1.0 (January 1994)
  - Python 2 (October 2000)
    - Python 2.7 (July 2010) – latest 2.x version († 2020)

A Glimpse at Python History

- invented in the beginning of 1990s by Guido van Rossum

- its name stems from "Monty Python's Flying Circus"
- version history:
  - Python 1.0 (January 1994)
  - Python 2 (October 2000)
    - Python 2.7 (July 2010) – latest 2.x version († 2020)
  - Python 3 (December 2008)
    - Python 3.7 (June 2018) – latest 3.x version

Diving Into Python

- interactive shell

```
$ python
Python 3.7.2 (default, Jan 10 2019, 23:51:51)
>>> print('Hello, world!')
Hello, world!
```

Diving Into Python

- interactive shell

```
$ python
Python 3.7.2 (default, Jan 10 2019, 23:51:51)
>>> print('Hello, world!')
Hello, world!
```

- running from source

```
# In file hello.py:
print('Hello, world!')

$ python hello.py
Hello, world!
```

Diving Into Python

- interactive shell

```
$ python
Python 3.7.2 (default, Jan 10 2019, 23:51:51)
>>> print('Hello, world!')
Hello, world!
```

- running from source

```
# In file hello.py:
print('Hello, world!')

$ python hello.py
Hello, world!
```

- combination

```
$ python -i hello.py
Hello, world!
>>>
```

Built-In Primitive Data Types

- NoneType

```
None
```

Built-In Primitive Data Types

- NoneType

```
None
```

- bool

```
True, False
```

Built-In Primitive Data Types

- NoneType

```
None
```

- bool

```
True, False
```

- int

```
-1024, 0, 178212237348573485387462734645 45
```

Built-In Primitive Data Types

- NoneType

```
None
```

- bool

```
True, False
```

- int

```
-1024, 0, 178212237348573485387462734645 45
```

- float

```
0.125, 1e200, float('inf'), float('nan')
```

Built-In Primitive Data Types

- NoneType

```
None
```

- bool

```
True, False
```

- int

```
-1024, 0, 178212237348573485387462734645 45
```

- float

```
0.125, 1e200, float('inf'), float('nan')
```

- complex

```
2 + 3j
```

Built-In Primitive Data Types

- NoneType

```
None
```

- bool

```
True, False
```

- int

```
-1024, 0, 178212237348573485387462734645 45
```

- float

```
0.125, 1e200, float('inf'), float('nan')
```

- complex

```
2 + 3j
```

- str

```
'Do you like jalapeño peppers?'
```

Built-In Primitive Data Types

- NoneType

```
None
```

- bool

```
True, False
```

- int

```
-1024, 0, 178212237348573485387462734645 45
```

- float

```
0.125, 1e200, float('inf'), float('nan')
```

- complex

```
2 + 3j
```

- str

```
'Do you like jalapeño peppers?'
```

- bytes

```
b'\x68\x65\x6c\x6c\x6f'
```

Intermezzo: Encodings

- character set vs encoding

Intermezzo: Encodings

- character set vs encoding
- single-byte vs multi-byte

Intermezzo: Encodings

- character set vs encoding
- single-byte vs multi-byte
- Unicode vs UTF-8, UTF-16, UTF-32

Intermezzo: Encodings

- character set vs encoding
- single-byte vs multi-byte
- Unicode vs UTF-8, UTF-16, UTF-32
- `str` vs `bytes`

Intermezzo: Encodings

- character set vs encoding
- single-byte vs multi-byte
- Unicode vs UTF-8, UTF-16, UTF-32
- `str` vs `bytes`

https://cs-blog.petrzemek.net/2015-08-09-znakova-sada-vs-kodovani

Built-In Collection Types

- list

```
[1, 2.0, 'hey!', None]
```

Built-In Collection Types

- list

```
[1, 2.0, 'hey!', None]
```

- tuple

```
('Cabernet Sauvignon', 1995)
```

Built-In Collection Types

- list

```
[1, 2.0, 'hey!', None]
```

- tuple

```
('Cabernet Sauvignon', 1995)
```

- set

```
{1, 2, 3, 4, 5}
```

Built-In Collection Types

- list

```
[1, 2.0, 'hey!', None]
```

- tuple

```
('Cabernet Sauvignon', 1995)
```

- set

```
{1, 2, 3, 4, 5}
```

- dict

```
{
    'John': 2.5,
    'Paul': 1.5,
    'Laura': 1,
}
```

Variables and Bindings

- name binding (we attach a name to an object)

Variables and Bindings

- name binding (we attach a name to an object)
- dynamic typing

Variables and Bindings

- name binding (we attach a name to an object)
- dynamic typing
- no explicit declarations until Python 3.5 (*type hints*)

Variables and Bindings

- name binding (we attach a name to an object)
- dynamic typing
- no explicit declarations until Python 3.5 (*type hints*)

```
>>>  x = 1                    # x --> 1
```

Variables and Bindings

- name binding (we attach a name to an object)
- dynamic typing
- no explicit declarations until Python 3.5 (*type hints*)

```
>>>  x = 1                    # x --> 1
>>>  x = 'hi there'           # x --> 'hi there'
```

Variables and Bindings

- name binding (we attach a name to an object)
- dynamic typing
- no explicit declarations until Python 3.5 (*type hints*)

```
>>>  x = 1                    # x --> 1
>>>  x = 'hi there'           # x --> 'hi there'

>>>  a = [1, 2]               # a --> [1, 2]
```

Variables and Bindings

- name binding (we attach a name to an object)
- dynamic typing
- no explicit declarations until Python 3.5 (*type hints*)

```
>>>  x = 1                    # x --> 1
>>>  x = 'hi there'           # x --> 'hi there'

>>>  a = [1, 2]               # a --> [1, 2]
>>>  b = a                    # a --> [1, 2] <-- b
```

Variables and Bindings

- name binding (we attach a name to an object)
- dynamic typing
- no explicit declarations until Python 3.5 (*type hints*)

```
>>>  x = 1                    # x --> 1
>>>  x = 'hi there'           # x --> 'hi there'

>>>  a = [1, 2]               # a --> [1, 2]
>>>  b = a                    # a --> [1, 2] <-- b
>>>  a.append(3)              # a --> [1, 2, 3] <-- b
```

Variables and Bindings

- name binding (we attach a name to an object)
- dynamic typing
- no explicit declarations until Python 3.5 (*type hints*)

```
>>>  x = 1                    # x --> 1
>>>  x = 'hi there'           # x --> 'hi there'

>>>  a = [1, 2]               # a --> [1, 2]
>>>  b = a                    # a --> [1, 2] <-- b
>>>  a.append(3)              # a --> [1, 2, 3] <-- b
>>>  a
[1, 2, 3]
>>>  b
[1, 2, 3]
```

Variables and Bindings

- name binding (we attach a name to an object)
- dynamic typing
- no explicit declarations until Python 3.5 (*type hints*)

```
>>>  x = 1                    # x --> 1
>>>  x = 'hi there'           # x --> 'hi there'

>>>  a = [1, 2]               # a --> [1, 2]
>>>  b = a                    # a --> [1, 2] <-- b
>>>  a.append(3)              # a --> [1, 2, 3] <-- b
>>>  a
[1, 2, 3]
>>>  b
[1, 2, 3]
>>>  b = [4]                  # a --> [1, 2, 3]; b --> [4]
```

Operations

arithmetic `+` `-` `*` `/` `//` `%` `**` `@`

Operations

arithmetic `+ - * / // % ** @`

comparison `== != < > <= >=`

Operations

arithmetic	`+ - * / // % ** @`
comparison	`== != < > <= >=`
bitwise	`<< >> | & ^ ~`

Operations

arithmetic	`+ - * / // % ** @`
comparison	`== != < > <= >=`
bitwise	`<< >> | & ^ ~`
indexing	`[]`

Operations

arithmetic	`+ - * / // % ** @`
comparison	`== != < > <= >=`
bitwise	`<< >> | & ^ ~`
indexing	`[]`
slicing	`[:]`

Operations

arithmetic	`+ - * / // % ** @`
comparison	`== != < > <= >=`
bitwise	`<< >> | & ^ ~`
indexing	`[]`
slicing	`[:]`
call	`()`

Operations

arithmetic	`+ - * / // % ** @`
comparison	`== != < > <= >=`
bitwise	`<< >> | & ^ ~`
indexing	`[]`
slicing	`[:]`
call	`()`
logical	and or not

Operations

arithmetic	`+ - * / // % ** @`
comparison	`== != < > <= >=`
bitwise	`<< >> | & ^ ~`
indexing	`[]`
slicing	`[:]`
call	`()`
logical	and or not
assignment	`= += -= *= /= //= %= **=` ...

Operations

arithmetic	`+ - * / // % ** @`
comparison	`== != < > <= >=`
bitwise	`<< >> | & ^ ~`
indexing	`[]`
slicing	`[:]`
call	`()`
logical	and or not
assignment	`= += -= *= /= //= %= **=` ...
other	in is

Basic Statements

`=` assignment statements

```
x = 1
x += 41
```

Basic Statements

`=` assignment statements

```
x = 1
x += 41
```

(expr) expression statements

```
print('My name is', name)
```

Basic Statements

`=` assignment statements

```
x = 1
x += 41
```

(expr) expression statements

```
print('My name is', name)
```

if conditional execution

```
if x > 10:
    x = 10
elif x < 5:
    x = 5
else:
    print('error')
```

Basic Statements (Continued)

for

traversing collections

```
for color in ['red', 'green', 'blue']:
    print(color)
```

Basic Statements (Continued)

for traversing collections

```
for color in ['red', 'green', 'blue']:
    print(color)
```

while repeated execution

```
while x > 0:
    print(x)
    x -= 1
```

Basic Statements (Continued)

for traversing collections

```
for color in ['red', 'green', 'blue']:
    print(color)
```

while repeated execution

```
while x > 0:
    print(x)
    x -= 1
```

break breaking from a loop

Basic Statements (Continued)

for traversing collections

```
for color in ['red', 'green', 'blue']:
    print(color)
```

while repeated execution

```
while x > 0:
    print(x)
    x -= 1
```

break breaking from a loop

continue continuing with the next cycle of a loop

Basic Statements (Continued)

for traversing collections

```
for color in ['red', 'green', 'blue']:
    print(color)
```

while repeated execution

```
while x > 0:
    print(x)
    x -= 1
```

break breaking from a loop

continue continuing with the next cycle of a loop

assert assertions

Basic Statements (Continued)

for traversing collections

```
for color in ['red', 'green', 'blue']:
    print(color)
```

while repeated execution

```
while x > 0:
    print(x)
    x -= 1
```

break breaking from a loop

continue continuing with the next cycle of a loop

assert assertions

return returning from a function

Basic Statements (Continued)

for — traversing collections

```
for color in ['red', 'green', 'blue']:
    print(color)
```

while — repeated execution

```
while x > 0:
    print(x)
    x -= 1
```

break — breaking from a loop

continue — continuing with the next cycle of a loop

assert — assertions

return — returning from a function

pass — does nothing

Functions

```
def factorial(n):
    """Returns the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

x = factorial(5)  # 120
```

Functions

```
def factorial(n):
    """Returns the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

x = factorial(5)  # 120
```

- first-class objects

Functions

```
def factorial(n):
    """Returns the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

x = factorial(5)  # 120
```

- first-class objects
- can be nested

Functions

```
def factorial(n):
    """Returns the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

x = factorial(5)  # 120
```

- first-class objects
- can be nested
- default arguments

Functions

```
def factorial(n):
    """Returns the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

x = factorial(5)  # 120
```

- first-class objects
- can be nested
- default arguments
- keyword arguments

Functions

```
def factorial(n):
    """Returns the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

x = factorial(5)  # 120
```

- first-class objects
- can be nested
- default arguments
- keyword arguments
- variable-length arguments

Scoping

Scoping

```
...  # A
def foo():
    ...  # B
    def bar():
        ...  # C
        while cond:
            ... # D
            print(x)
```

Scoping

```
...   # A
def foo():
    ...   # B
    def bar():
        ...   # C
        while cond:
            ... # D
            print(x)
```

- lexical scoping

Scoping

```
...  # A
def foo():
    ...  # B
    def bar():
        ...  # C
        while cond:
            ... # D
            print(x)
```

- lexical scoping
- LEGB: a concise rule for scope resolution
  1. **L**ocal
  2. **E**nclosing
  3. **G**lobal
  4. **B**uilt-in

Scoping

```
...  # A
def foo():
    ...  # B
    def bar():
        ...  # C
        while cond:
            ... # D
            print(x)
```

- lexical scoping
- LEGB: a concise rule for scope resolution
  1. **L**ocal
  2. **E**nclosing
  3. **G**lobal
  4. **B**uilt-in
- `if`, `for`, etc. do not introduce a new scope

Scoping

```
...  # A
def foo():
    ...  # B
    def bar():
        ...  # C
        while cond:
            ... # D
            print(x)
```

- lexical scoping
- LEGB: a concise rule for scope resolution
  1. **L**ocal
  2. **E**nclosing
  3. **G**lobal
  4. **B**uilt-in
- `if`, `for`, etc. do not introduce a new scope
- explicit declarations via `global` and `nonlocal`

Lifetimes

Lifetimes

- global variables exist until program exits

Lifetimes

- global variables exist until program exits
- local variables exist until function exits

Lifetimes

- global variables exist until program exits
- local variables exist until function exits
- explicit deletion via `del`

Namespaces, Modules, and Packages

```
# Example of a custom package:

network/
    __init__.py
    socket.py
    http/
        __init__.py
        request.py
        response.py
        ...
    bittorrent/
        __init__.py
        torrent.py
        bencoding.py
        ...
    ...
```

Namespaces, Modules, and Packages

```
# Example of a custom package:

network/
    __init__.py
    socket.py
    http/
        __init__.py
        request.py
        response.py
        ...
    bittorrent/
        __init__.py
        torrent.py
        bencoding.py
        ...
    ...

from network.http.request import Request
```

Imports

```
# Import a single module.
import time
```

Imports

```
# Import a single module.
import time

# Import multiple modules at once.
import os, re, sys
```

Imports

```
# Import a single module.
import time

# Import multiple modules at once.
import os, re, sys

# Import under a different name.
import multiprocessing as mp
```

Imports

```
# Import a single module.
import time

# Import multiple modules at once.
import os, re, sys

# Import under a different name.
import multiprocessing as mp

# Import a single item from a module.
from threading import Thread
```

Imports

```
# Import a single module.
import time

# Import multiple modules at once.
import os, re, sys

# Import under a different name.
import multiprocessing as mp

# Import a single item from a module.
from threading import Thread

# Import multiple items from a module.
from collections import namedtuple, defaultdict
```

Imports

```
# Import a single module.
import time

# Import multiple modules at once.
import os, re, sys

# Import under a different name.
import multiprocessing as mp

# Import a single item from a module.
from threading import Thread

# Import multiple items from a module.
from collections import namedtuple, defaultdict

# Import everything from the given module.
# (Use with caution!)
from email import *
```

Object-Oriented Programming

```
from math import sqrt

class Point:
    """Representation of a point in 2D space."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        return sqrt((other.x - self.x) ** 2 +
                    (other.y - self.y) ** 2)

a = Point(1, 2)
b = Point(3, 4)
print(a.distance(b))  # 2.8284271247461903
```

Object-Oriented Programming (Basics)

- instance creation and initialization

Object-Oriented Programming (Basics)

- instance creation and initialization
- methods versus functions

Object-Oriented Programming (Basics)

- instance creation and initialization
- methods versus functions
- classes are first-class objects

Object-Oriented Programming (Basics)

- instance creation and initialization
- methods versus functions
- classes are first-class objects
- everything is public

Object-Oriented Programming (Basics)

- instance creation and initialization
- methods versus functions
- classes are first-class objects
- everything is public
- everything can be overridden

Object-Oriented Programming (Basics)

- instance creation and initialization
- methods versus functions
- classes are first-class objects
- everything is public
- everything can be overridden
- each class automatically inherits from `object`

Object-Oriented Programming (Basics)

- instance creation and initialization
- methods versus functions
- classes are first-class objects
- everything is public
- everything can be overridden
- each class automatically inherits from `object`
- multiple inheritance, method resolution order (MRO)

Object-Oriented Programming (Basics)

- instance creation and initialization
- methods versus functions
- classes are first-class objects
- everything is public
- everything can be overridden
- each class automatically inherits from `object`
- multiple inheritance, method resolution order (MRO)
- calling base-class methods

Object-Oriented Programming (Basics)

- instance creation and initialization
- methods versus functions
- classes are first-class objects
- everything is public
- everything can be overridden
- each class automatically inherits from `object`
- multiple inheritance, method resolution order (MRO)
- calling base-class methods
- instance variables vs class variables

Object-Oriented Programming (Basics)

- instance creation and initialization
- methods versus functions
- classes are first-class objects
- everything is public
- everything can be overridden
- each class automatically inherits from `object`
- multiple inheritance, method resolution order (MRO)
- calling base-class methods
- instance variables vs class variables
- instance methods vs class methods vs static methods

Object-Oriented Programming (Basics)

- instance creation and initialization
- methods versus functions
- classes are first-class objects
- everything is public
- everything can be overridden
- each class automatically inherits from `object`
- multiple inheritance, method resolution order (MRO)
- calling base-class methods
- instance variables vs class variables
- instance methods vs class methods vs static methods
- properties

Object-Oriented Programming (Advanced)

- instance creation in detail (`__new__()`, `__init__()`)

Object-Oriented Programming (Advanced)

- instance creation in detail (`__new__()`, `__init__()`)
- instance memory layout (`__dict__`, `__slots__`)

Object-Oriented Programming (Advanced)

- instance creation in detail (`__new__()`, `__init__()`)
- instance memory layout (`__dict__`, `__slots__`)
- "internal" (`_`) and pseudo-private (`__`) attributes

Object-Oriented Programming (Advanced)

- instance creation in detail (`__new__()`, `__init__()`)
- instance memory layout (`__dict__`, `__slots__`)
- "internal" (`_`) and pseudo-private (`__`) attributes
- special methods (`__$method__()`), operator overloading

Object-Oriented Programming (Advanced)

- instance creation in detail (`__new__()`, `__init__()`)
- instance memory layout (`__dict__`, `__slots__`)
- "internal" (`_`) and pseudo-private (`__`) attributes
- special methods (`__$method__()`), operator overloading
- cooperative multiple inheritance, mixins, `super()`

Object-Oriented Programming (Advanced)

- instance creation in detail (`__new__()`, `__init__()`)
- instance memory layout (`__dict__`, `__slots__`)
- "internal" (`_`) and pseudo-private (`__`) attributes
- special methods (`__$method__()`), operator overloading
- cooperative multiple inheritance, mixins, `super()`
- instance finalization (`__del__()`)

Object-Oriented Programming (Advanced)

- instance creation in detail (`__new__()`, `__init__()`)
- instance memory layout (`__dict__`, `__slots__`)
- "internal" (`_`) and pseudo-private (`__`) attributes
- special methods (`__$method__()`), operator overloading
- cooperative multiple inheritance, mixins, `super()`
- instance finalization (`__del__()`)
- hooking into attribute lookup (`__getattr[ibute]__()`)

Object-Oriented Programming (Advanced)

- instance creation in detail (`__new__()`, `__init__()`)
- instance memory layout (`__dict__`, `__slots__`)
- "internal" (`_`) and pseudo-private (`__`) attributes
- special methods (`__$method__()`), operator overloading
- cooperative multiple inheritance, mixins, `super()`
- instance finalization (`__del__()`)
- hooking into attribute lookup (`__getattr[ibute]__()`)
- protocols, duck typing

Object-Oriented Programming (Advanced)

- instance creation in detail (`__new__()`, `__init__()`)
- instance memory layout (`__dict__`, `__slots__`)
- "internal" (`_`) and pseudo-private (`__`) attributes
- special methods (`__$method__()`), operator overloading
- cooperative multiple inheritance, mixins, `super()`
- instance finalization (`__del__()`)
- hooking into attribute lookup (`__getattr[ibute]__()`)
- protocols, duck typing
- interfaces, abstract base classes (`abc`)

Object-Oriented Programming (Advanced)

- instance creation in detail (`__new__()`, `__init__()`)
- instance memory layout (`__dict__`, `__slots__`)
- "internal" (`_`) and pseudo-private (`__`) attributes
- special methods (`__$method__()`), operator overloading
- cooperative multiple inheritance, mixins, `super()`
- instance finalization (`__del__()`)
- hooking into attribute lookup (`__getattr[ibute]__()`)
- protocols, duck typing
- interfaces, abstract base classes (`abc`)
- classes can be created and extended during runtime

Object-Oriented Programming (Advanced)

- instance creation in detail (`__new__()`, `__init__()`)
- instance memory layout (`__dict__`, `__slots__`)
- "internal" (`_`) and pseudo-private (`__`) attributes
- special methods (`__$method__()`), operator overloading
- cooperative multiple inheritance, mixins, `super()`
- instance finalization (`__del__()`)
- hooking into attribute lookup (`__getattr[ibute]__()`)
- protocols, duck typing
- interfaces, abstract base classes (`abc`)
- classes can be created and extended during runtime
- classes are instances of *metaclasses*

Error Handling and Exceptions

```
# Raising an exception:
raise IOError('not enough space')
```

Error Handling and Exceptions

```
# Raising an exception:
raise IOError('not enough space')


# Exception handling:
try:
    # code
```

Error Handling and Exceptions

```
# Raising an exception:
raise IOError('not enough space')


# Exception handling:
try:
    # code
except IOError as ex:
    # handle a specific exception
```

Error Handling and Exceptions

```
# Raising an exception:
raise IOError('not enough space')


# Exception handling:
try:
    # code
except IOError as ex:
    # handle a specific exception
except:
    # handle all other exceptions
```

Error Handling and Exceptions

```
# Raising an exception:
raise IOError('not enough space')


# Exception handling:
try:
    # code
except IOError as ex:
    # handle a specific exception
except:
    # handle all other exceptions
else:
    # no exception was raised
```

Error Handling and Exceptions

```
# Raising an exception:
raise IOError('not enough space')


# Exception handling:
try:
    # code
except IOError as ex:
    # handle a specific exception
except:
    # handle all other exceptions
else:
    # no exception was raised
finally:
    # clean-up actions, always executed
```

Exception-Safe Resource Management

```
# Bad:
f = open('file.txt', 'r')
contents = f.read()
f.close()
```

Exception-Safe Resource Management

```
# Bad:
f = open('file.txt', 'r')
contents = f.read()
f.close()

# Better:
f = open('file.txt', 'r')
try:
    contents = f.read()
finally:
    f.close()
```

Exception-Safe Resource Management

```
# Bad:
f = open('file.txt', 'r')
contents = f.read()
f.close()

# Better:
f = open('file.txt', 'r')
try:
    contents = f.read()
finally:
    f.close()

# The best:
with open('file.txt', 'r') as f:
    contents = f.read()
```

Exception-Safe Resource Management

```
# Bad:
f = open('file.txt', 'r')
contents = f.read()
f.close()

# Better:
f = open('file.txt', 'r')
try:
    contents = f.read()
finally:
    f.close()

# The best:
with open('file.txt', 'r') as f:
    contents = f.read()
```

https://cs-blog.petrzemek.net/2013-11-17-jeste-jednou-a-lepe-prace-se-souborem-v-pythonu

Intermezzo: Text vs Binary Files

- text vs binary mode

```
with open(file_path, 'r') as f:
    text = f.read()

with open(file_path, 'rb') as f:
    data = f.read()
```

Intermezzo: Text vs Binary Files

- text vs binary mode

```
with open(file_path, 'r') as f:
    text = f.read()

with open(file_path, 'rb') as f:
    data = f.read()
```

- differences between text and binary modes in Python:

Intermezzo: Text vs Binary Files

- text vs binary mode

```
with open(file_path, 'r') as f:
    text = f.read()

with open(file_path, 'rb') as f:
    data = f.read()
```

- differences between text and binary modes in Python:
  1. decoding

Intermezzo: Text vs Binary Files

- text vs binary mode

```
with open(file_path, 'r') as f:
    text = f.read()

with open(file_path, 'rb') as f:
    data = f.read()
```

- differences between text and binary modes in Python:
  1. decoding
  2. end-of-line conversions

Intermezzo: Text vs Binary Files

- text vs binary mode

```
with open(file_path, 'r') as f:
    text = f.read()

with open(file_path, 'rb') as f:
    data = f.read()
```

- differences between text and binary modes in Python:
  1. decoding
  2. end-of-line conversions
  3. buffering

Intermezzo: Text vs Binary Files

- text vs binary mode

```
with open(file_path, 'r') as f:
    text = f.read()

with open(file_path, 'rb') as f:
    data = f.read()
```

- differences between text and binary modes in Python:
  1. decoding
  2. end-of-line conversions
  3. buffering

https://cs-blog.petrzemek.net/2015-08-26-textove-vs-binarni-soubory

Some Cool Language Features

- string formatting (*f-strings*, Python 3.6)

```
name = 'Joe'
item = 'bike'
print(f'Hey {name}, where is my {item}?')
```

Some Cool Language Features

- string formatting (*f-strings*, Python 3.6)

```
name = 'Joe'
item = 'bike'
print(f'Hey {name}, where is my {item}?')
```

- anonymous functions

```
people.sort(key=lambda person: person.name)
```

Some Cool Language Features

- string formatting (*f-strings*, Python 3.6)

```
name = 'Joe'
item = 'bike'
print(f'Hey {name}, where is my {item}?')
```

- anonymous functions

```
people.sort(key=lambda person: person.name)
```

- list/set/dict comprehensions

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = [x ** 2 for x in list if x % 2 == 0]
# [4, 16, 36, 64, 100]
```

Some Cool Language Features

- string formatting (*f-strings*, Python 3.6)

```
name = 'Joe'
item = 'bike'
print(f'Hey {name}, where is my {item}?')
```

- anonymous functions

```
people.sort(key=lambda person: person.name)
```

- list/set/dict comprehensions

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = [x ** 2 for x in list if x % 2 == 0]
# [4, 16, 36, 64, 100]
```

- conditional expressions

```
cost = 'cheap' if price <= 100 else 'expensive'
```

Some Cool Language Features (Continued)

- `eval()` and `exec()`

```
a = eval('1 + 3')          # a = 4
exec('b = [1, 2, 3]')      # b = [1, 2, 3]
```

Some Cool Language Features (Continued)

- `eval()` and `exec()`

```
a = eval('1 + 3')          # a = 4
exec('b = [1, 2, 3]')      # b = [1, 2, 3]
```

- dynamic typing

```
def print_all(col):
    for i in col:
        print(i)

print_all([1, 2, 3])
print_all(('a', 'b', 'c'))
```

Some Cool Language Features (Continued)

- `eval()` and `exec()`

```
a = eval('1 + 3')          # a = 4
exec('b = [1, 2, 3]')      # b = [1, 2, 3]
```

- dynamic typing

```
def print_all(col):
    for i in col:
        print(i)

print_all([1, 2, 3])
print_all(('a', 'b', 'c'))
```

- `enumerate()`

```
for i, person in enumerate(people):
    print(i, ':', person)
```

Some Cool Language Features (Continued)

- chained comparisons

```
if 1 < x < 5:
    # ...
```

Some Cool Language Features (Continued)

- chained comparisons

```
if 1 < x < 5:
    # ...
```

- digits separator (Python 3.6)

```
1_483_349_803
```

Some Cool Language Features (Continued)

- chained comparisons

```
if 1 < x < 5:
    # ...
```

- digits separator (Python 3.6)

```
1_483_349_803
```

- tuple unpacking

```
head, *middle, tail = [1, 2, 3, 4, 5]
```

Some Cool Language Features (Continued)

- generators

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib = fibonacci()
next(fib)   # 0
next(fib)   # 1
next(fib)   # 1
next(fib)   # 2
next(fib)   # 3
next(fib)   # 5
next(fib)   # 8
```

Weird Language Features

- `for` with `else`

```
for item in some_list:
    if item == 5:
        break
else:
    print("not found")
```

Weird Language Features

- `for` with `else`

```
for item in some_list:
    if item == 5:
        break
else:
    print("not found")
```

- mutating default arguments

```
def foo(x=[]):
    x.append(4)
    return x

print(foo([1, 2, 3]))  # [1, 2, 3, 4]
print(foo())           # [4]
print(foo())           # [4, 4]
```

Weird Language Features

- `for` with `else`

```
for item in some_list:
    if item == 5:
        break
else:
    print("not found")
```

- mutating default arguments

```
def foo(x=[]):
    x.append(4)
    return x

print(foo([1, 2, 3]))  # [1, 2, 3, 4]
print(foo())           # [4]
print(foo())           # [4, 4]
```

- non-ASCII identifiers

```
π = 3.1415
```

What We Have Skipped

- metaclasses
- decorators
- descriptors
- context managers
- threading
- multiprocessing
- asynchronous I/O
- coroutines
- annotations (including type hints)
- ... and more ...

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)
- concurrency (`threading`, `multiprocessing`, `asyncio`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)
- concurrency (`threading`, `multiprocessing`, `asyncio`)
- math (`math`, `decimal`, `fractions`, `statistics`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)
- concurrency (`threading`, `multiprocessing`, `asyncio`)
- math (`math`, `decimal`, `fractions`, `statistics`)
- operating system and filesystem (`os`, `shutil`, `tempfile`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)
- concurrency (`threading`, `multiprocessing`, `asyncio`)
- math (`math`, `decimal`, `fractions`, `statistics`)
- operating system and filesystem (`os`, `shutil`, `tempfile`)
- IPC and networking (`signal`, `mmap`, `select`, `socket`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)
- concurrency (`threading`, `multiprocessing`, `asyncio`)
- math (`math`, `decimal`, `fractions`, `statistics`)
- operating system and filesystem (`os`, `shutil`, `tempfile`)
- IPC and networking (`signal`, `mmap`, `select`, `socket`)
- Internet protocols (`urllib`, `email`, `smtplib`, `ipaddress`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)
- concurrency (`threading`, `multiprocessing`, `asyncio`)
- math (`math`, `decimal`, `fractions`, `statistics`)
- operating system and filesystem (`os`, `shutil`, `tempfile`)
- IPC and networking (`signal`, `mmap`, `select`, `socket`)
- Internet protocols (`urllib`, `email`, `smtplib`, `ipaddress`)
- compression (`zipfile`, `tarfile`, `gzip`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)
- concurrency (`threading`, `multiprocessing`, `asyncio`)
- math (`math`, `decimal`, `fractions`, `statistics`)
- operating system and filesystem (`os`, `shutil`, `tempfile`)
- IPC and networking (`signal`, `mmap`, `select`, `socket`)
- Internet protocols (`urllib`, `email`, `smtplib`, `ipaddress`)
- compression (`zipfile`, `tarfile`, `gzip`)
- cryptography (`hashlib`, `secrets`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)
- concurrency (`threading`, `multiprocessing`, `asyncio`)
- math (`math`, `decimal`, `fractions`, `statistics`)
- operating system and filesystem (`os`, `shutil`, `tempfile`)
- IPC and networking (`signal`, `mmap`, `select`, `socket`)
- Internet protocols (`urllib`, `email`, `smtplib`, `ipaddress`)
- compression (`zipfile`, `tarfile`, `gzip`)
- cryptography (`hashlib`, `secrets`)
- functional-like programming (`itertools`, `functools`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)
- concurrency (`threading`, `multiprocessing`, `asyncio`)
- math (`math`, `decimal`, `fractions`, `statistics`)
- operating system and filesystem (`os`, `shutil`, `tempfile`)
- IPC and networking (`signal`, `mmap`, `select`, `socket`)
- Internet protocols (`urllib`, `email`, `smtplib`, `ipaddress`)
- compression (`zipfile`, `tarfile`, `gzip`)
- cryptography (`hashlib`, `secrets`)
- functional-like programming (`itertools`, `functools`)
- development (`unittest`, `doctest`, `venv`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)
- concurrency (`threading`, `multiprocessing`, `asyncio`)
- math (`math`, `decimal`, `fractions`, `statistics`)
- operating system and filesystem (`os`, `shutil`, `tempfile`)
- IPC and networking (`signal`, `mmap`, `select`, `socket`)
- Internet protocols (`urllib`, `email`, `smtplib`, `ipaddress`)
- compression (`zipfile`, `tarfile`, `gzip`)
- cryptography (`hashlib`, `secrets`)
- functional-like programming (`itertools`, `functools`)
- development (`unittest`, `doctest`, `venv`)
- debugging and profiling (`pdb`, `timeit`, `dis`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)
- concurrency (`threading`, `multiprocessing`, `asyncio`)
- math (`math`, `decimal`, `fractions`, `statistics`)
- operating system and filesystem (`os`, `shutil`, `tempfile`)
- IPC and networking (`signal`, `mmap`, `select`, `socket`)
- Internet protocols (`urllib`, `email`, `smtplib`, `ipaddress`)
- compression (`zipfile`, `tarfile`, `gzip`)
- cryptography (`hashlib`, `secrets`)
- functional-like programming (`itertools`, `functools`)
- development (`unittest`, `doctest`, `venv`)
- debugging and profiling (`pdb`, `timeit`, `dis`)
- other (`logging`, `argparse`, `ctypes`)

A Brief Tour of the Standard Library

- text processing (`re`, `json`, `xml`, `csv`, `base64`)
- data types (`datetime`, `collections`, `dataclasses`)
- concurrency (`threading`, `multiprocessing`, `asyncio`)
- math (`math`, `decimal`, `fractions`, `statistics`)
- operating system and filesystem (`os`, `shutil`, `tempfile`)
- IPC and networking (`signal`, `mmap`, `select`, `socket`)
- Internet protocols (`urllib`, `email`, `smtplib`, `ipaddress`)
- compression (`zipfile`, `tarfile`, `gzip`)
- cryptography (`hashlib`, `secrets`)
- functional-like programming (`itertools`, `functools`)
- development (`unittest`, `doctest`, `venv`)
- debugging and profiling (`pdb`, `timeit`, `dis`)
- other (`logging`, `argparse`, `ctypes`)
- ...

Some Other Interesting Libraries and Projects

- `pip` (installation of Python packages)

Some Other Interesting Libraries and Projects

- `pip` (installation of Python packages)
- `requests` (HTTP for humans)

Some Other Interesting Libraries and Projects

- `pip` (installation of Python packages)
- `requests` (HTTP for humans)
- `sphinx` (documentation)

Some Other Interesting Libraries and Projects

- `pip` (installation of Python packages)
- `requests` (HTTP for humans)
- `sphinx` (documentation)
- `sqlalchemy` (database toolkit)

Some Other Interesting Libraries and Projects

- `pip` (installation of Python packages)
- `requests` (HTTP for humans)
- `sphinx` (documentation)
- `sqlalchemy` (database toolkit)
- `numpy`, `scipy` (scientific computing)

Some Other Interesting Libraries and Projects

- `pip` (installation of Python packages)
- `requests` (HTTP for humans)
- `sphinx` (documentation)
- `sqlalchemy` (database toolkit)
- `numpy`, `scipy` (scientific computing)
- `django`, `flask` (web frameworks)

Some Other Interesting Libraries and Projects

- `pip` (installation of Python packages)
- `requests` (HTTP for humans)
- `sphinx` (documentation)
- `sqlalchemy` (database toolkit)
- `numpy`, `scipy` (scientific computing)
- `django`, `flask` (web frameworks)
- `coverage` (code coverage)

Some Other Interesting Libraries and Projects

- `pip` (installation of Python packages)
- `requests` (HTTP for humans)
- `sphinx` (documentation)
- `sqlalchemy` (database toolkit)
- `numpy`, `scipy` (scientific computing)
- `django`, `flask` (web frameworks)
- `coverage` (code coverage)
- `ply` (Python Lex and Yacc)

Some Other Interesting Libraries and Projects

- `pip` (installation of Python packages)
- `requests` (HTTP for humans)
- `sphinx` (documentation)
- `sqlalchemy` (database toolkit)
- `numpy`, `scipy` (scientific computing)
- `django`, `flask` (web frameworks)
- `coverage` (code coverage)
- `ply` (Python Lex and Yacc)
- `matplotlib` (2D plotting)

Some Other Interesting Libraries and Projects

- `pip` (installation of Python packages)
- `requests` (HTTP for humans)
- `sphinx` (documentation)
- `sqlalchemy` (database toolkit)
- `numpy`, `scipy` (scientific computing)
- `django`, `flask` (web frameworks)
- `coverage` (code coverage)
- `ply` (Python Lex and Yacc)
- `matplotlib` (2D plotting)
- `pygal` (charting)

Some Other Interesting Libraries and Projects

- `pip` (installation of Python packages)
- `requests` (HTTP for humans)
- `sphinx` (documentation)
- `sqlalchemy` (database toolkit)
- `numpy`, `scipy` (scientific computing)
- `django`, `flask` (web frameworks)
- `coverage` (code coverage)
- `ply` (Python Lex and Yacc)
- `matplotlib` (2D plotting)
- `pygal` (charting)
- `pygame` (games)

Some Other Interesting Libraries and Projects

- `pip` (installation of Python packages)
- `requests` (HTTP for humans)
- `sphinx` (documentation)
- `sqlalchemy` (database toolkit)
- `numpy`, `scipy` (scientific computing)
- `django`, `flask` (web frameworks)
- `coverage` (code coverage)
- `ply` (Python Lex and Yacc)
- `matplotlib` (2D plotting)
- `pygal` (charting)
- `pygame` (games)
- `pyqt` (GUI)

Advantages of Python

+ clean and simple syntax

Advantages of Python

+ clean and simple syntax
+ easy to learn

Advantages of Python

+ clean and simple syntax
+ easy to learn
+ productivity (high-level constructs)

Advantages of Python

+ clean and simple syntax
+ easy to learn
+ productivity (high-level constructs)
+ powerful built-in types

Advantages of Python

+ clean and simple syntax
+ easy to learn
+ productivity (high-level constructs)
+ powerful built-in types
+ elegant and flexible module system

Advantages of Python

+ clean and simple syntax
+ easy to learn
+ productivity (high-level constructs)
+ powerful built-in types
+ elegant and flexible module system
+ excellent standard library

Advantages of Python

+ clean and simple syntax
+ easy to learn
+ productivity (high-level constructs)
+ powerful built-in types
+ elegant and flexible module system
+ excellent standard library
+ reflection

Advantages of Python

+ clean and simple syntax
+ easy to learn
+ productivity (high-level constructs)
+ powerful built-in types
+ elegant and flexible module system
+ excellent standard library
+ reflection
+ multiparadigm (procedural, object oriented)

Advantages of Python

+ clean and simple syntax
+ easy to learn
+ productivity (high-level constructs)
+ powerful built-in types
+ elegant and flexible module system
+ excellent standard library
+ reflection
+ multiparadigm (procedural, object oriented)
+ generic programming (duck typing)

Advantages of Python

+ clean and simple syntax
+ easy to learn
+ productivity (high-level constructs)
+ powerful built-in types
+ elegant and flexible module system
+ excellent standard library
+ reflection
+ multiparadigm (procedural, object oriented)
+ generic programming (duck typing)
+ widely used

Disadvantages of Python

- not very fast on computationally intensive operations

Disadvantages of Python

- not very fast on computationally intensive operations
- not for memory-intensive tasks

Disadvantages of Python

- not very fast on computationally intensive operations
- not for memory-intensive tasks
- limited parallelism with threads (Global Interpreter Lock)

Disadvantages of Python

- not very fast on computationally intensive operations
- not for memory-intensive tasks
- limited parallelism with threads (Global Interpreter Lock)
- limited notion of constness

Disadvantages of Python

- not very fast on computationally intensive operations
- not for memory-intensive tasks
- limited parallelism with threads (Global Interpreter Lock)
- limited notion of constness
- portable, but some parts are OS-specific

Disadvantages of Python

- not very fast on computationally intensive operations
- not for memory-intensive tasks
- limited parallelism with threads (Global Interpreter Lock)
- limited notion of constness
- portable, but some parts are OS-specific
- Python 2 vs 3 (incompatibilities)

Varying Opinions

+/- everything is public

Varying Opinions

+/- everything is public

+/- unsystematic documentation

Varying Opinions

+/- everything is public
+/- unsystematic documentation
+/- whitespace is significant

Varying Opinions

+/- everything is public
+/- unsystematic documentation
+/- whitespace is significant
+/- standardization

Varying Opinions

+/- everything is public
+/- unsystematic documentation
+/- whitespace is significant
+/- standardization
+/- supports “monkey patching”

Varying Opinions

+/- everything is public
+/- unsystematic documentation
+/- whitespace is significant
+/- standardization
+/- supports “monkey patching”
+/- not suitable for writing low-level code

Varying Opinions

+/- everything is public
+/- unsystematic documentation
+/- whitespace is significant
+/- standardization
+/- supports “monkey patching”
+/- not suitable for writing low-level code
+/- dynamic typing

Varying Opinions

+/- everything is public
+/- unsystematic documentation
+/- whitespace is significant
+/- standardization
+/- supports "monkey patching"
+/- not suitable for writing low-level code
+/- dynamic typing

https://cs-blog.petrzemek.net/2014-10-26-co-se-mi-nelibi-na-pythonu

Summary

- imperative language
- multiparadigm (procedural, object oriented)
- strongly typed
- dynamically typed
- interpreted (translated to internal representation)
- modularity is directly supported (packages, modules)

Where to Look for Further Information?

Python Programming Language – Official Website

https://www.python.org/

Python 3 Documentation

https://docs.python.org/3/

Official Python 3 Tutorial

https://docs.python.org/3/tutorial/

Dive into Python 3

http://www.diveintopython3.net/

Learning Python, 5th Edition (2013)

http://shop.oreilly.com/product/0636920028154.do

Fluent Python (2015)

http://shop.oreilly.com/product/0636920032519.do

Témata bakalářských prací v Avastu

- *Zpětný překlad kódu aplikací (dekompilace)*

  C++, Python – https://retdec.com/

Témata bakalářských prací v Avastu

- *Zpětný překlad kódu aplikací (dekompilace)*
  C++, Python – https://retdec.com/
- *Detekce vzorů založená na nástroji YARA*
  Python, C++, C

Témata bakalářských prací v Avastu

- *Zpětný překlad kódu aplikací (dekompilace)*
  C++, Python – https://retdec.com/
- *Detekce vzorů založená na nástroji YARA*
  Python, C++, C
- *Shluková analýza souborů*
  Python, C++

Témata bakalářských prací v Avastu

- *Zpětný překlad kódu aplikací (dekompilace)*
  C++, Python – https://retdec.com/
- *Detekce vzorů založená na nástroji YARA*
  Python, C++, C
- *Shluková analýza souborů*
  Python, C++
- *Honeypot pro zachytávání a inteligentní zpracování hrozeb*
  Python

Témata bakalářských prací v Avastu

- *Zpětný překlad kódu aplikací (dekompilace)*
  C++, Python – https://retdec.com/
- *Detekce vzorů založená na nástroji YARA*
  Python, C++, C
- *Shluková analýza souborů*
  Python, C++
- *Honeypot pro zachytávání a inteligentní zpracování hrozeb*
  Python

Kontaktní osoba: Lukáš Zobal (izobal@fit.vutbr.cz)