

```
Computer player with simple artificial intelligence.
```

```
def find_card(self, played_cards):
```

```
    """  
    Suche Karte, die auf der Hand zu spielen ist.  
    Search a card which can be played on the pile of played cards.  
    """
```

```
    for card in self.cards:  
        if card.is playable(played_cards):
```

```
def move(self, i):
```

```
    """  
    Ein Spielzug.  
    One move.  
    """
```

```
    card = self.find_card(i)
```

```
    if card:
```

```
        # karte gefunden / found a playable card  
        print("{} spielt/plays {}".format(self.name, card))  
        self.cards.remove(card)  
        played_cards.append(card)  
        if not self.cards:
```

```
import logging  
logging.basicConfig(filename='valrpc.log', level=logging.INFO)
```

```
class MyPyGame:
```

```
    def __init__(self):
```

```
        self.data = {}  
        self.dreary = dreary, which is considered better than dreary,  
        self.dreary = dreary, which is considered better than dreary,  
        self.dreary = dreary, which is considered better than dreary,
```

```
    def add(x, y):
```

```
        """  
        Adds two numbers.  
        """
```

```
        return x + y
```

```
    def surprise(self, i):
```

```
        """  
        Takes an integer between 0 and 5.  
        """
```

```
        return i
```

```
    def __str__(self):
```

```
        return "MyPyGame"
```

```
    def __repr__(self):
```

```
        return "MyPyGame"
```



python™

PYTHON Introduction to the Basics

March 2021 | S. Linner, M. Lischewski, M. Richerzhagen | Forschungszentrum Jülich

Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

What is Python?

Python: Dynamic programming language which supports several different programming paradigms:

- Procedural programming
- Object oriented programming
- Functional programming

Standard: Python byte code is executed in the Python interpreter (similar to Java)

→ **platform independent code**

Why Python?

- Extremely versatile language
 - Website development, data analysis, server maintenance, numerical analysis, ...
- Syntax is clear, easy to read and learn (almost pseudo code)
- Common language
- Intuitive object oriented programming
- Full modularity, hierarchical packages
- Comprehensive standard library for many tasks
- Big community
- Simply extendable via C/C++, wrapping of C/C++ libraries
- **Focus: Programming speed**

History

- Start implementation in December 1989 by Guido van Rossum (CWI)
- 16.10.2000: Python 2.0
 - Unicode support
 - Garbage collector
 - Development process more community oriented
- 3.12.2008: Python 3.0
 - Not 100% backwards compatible
- 2007 & 2010 most popular programming language (TIOBE Index)
- Recommendation for scientific programming (Nature News, NPG, 2015)
- Current version: Python 3.9.2
- Python2 is out of support!¹

¹<https://python3statement.org/>

Zen of Python

- 20 software principles that influence the design of Python:

- 1 Beautiful is better than ugly.
- 2 Explicit is better than implicit.
- 3 Simple is better than complex.
- 4 Complex is better than complicated.
- 5 Flat is better than nested.
- 6 Sparse is better than dense.
- 7 Readability counts.
- 8 Special cases aren't special enough to break the rules.
- 9 Although practicality beats purity.
- 10 Errors should never pass silently.
- 11 Unless explicitly silenced.
- 12 ...

Is Python fast enough?

- For user programs: Python is fast enough!
- Most parts of Python are written in C
- For compute intensive algorithms: Fortran, C, C++ might be better
- Performance-critical parts can be re-implemented in C/C++ if necessary
- First analyse, then optimise!

Hello World!

hello_world.py

```
#!/usr/bin/env python3  
  
# This is a commentary  
print("Hello world!")
```

```
$ python3 hello_world.py  
Hello world!  
$
```

```
$ chmod 755 hello_world.py  
$ ./hello_world.py  
Hello world!  
$
```

Hello User

hello_user.py

```
#!/usr/bin/env python3

name = input("What's your name? ")
print("Hello", name)
```

```
$ ./hello_user.py
What's your name? Rebecca
Hello Rebecca
$
```

Strong and Dynamic Typing

Strong Typing:

- Object is of exactly one type! A string is always a string, an integer always an integer
- Counterexamples: PHP, JavaScript, C: `char` can be interpreted as `short`, `void *` can be everything

Dynamic Typing:

- No variable declaration
- Variable names can be assigned to different data types in the course of a program
- An object's attributes are checked only at run time
- **Duck typing** (an object is defined by its methods and attributes)

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.²

²James Whitcomb Riley

Example: Strong and Dynamic Typing

types.py

```
#!/usr/bin/env python3
number = 3
print(number, type(number))
print(number + 42)
number = "3"
print(number, type(number))
print(number + 42)
```

```
3 <class 'int'>
45
3 <class 'str'>
Traceback (most recent call last):
  File "types.py", line 7, in <module>
    print(number + 42)
TypeError: can only concatenate str (not "int") to str
```

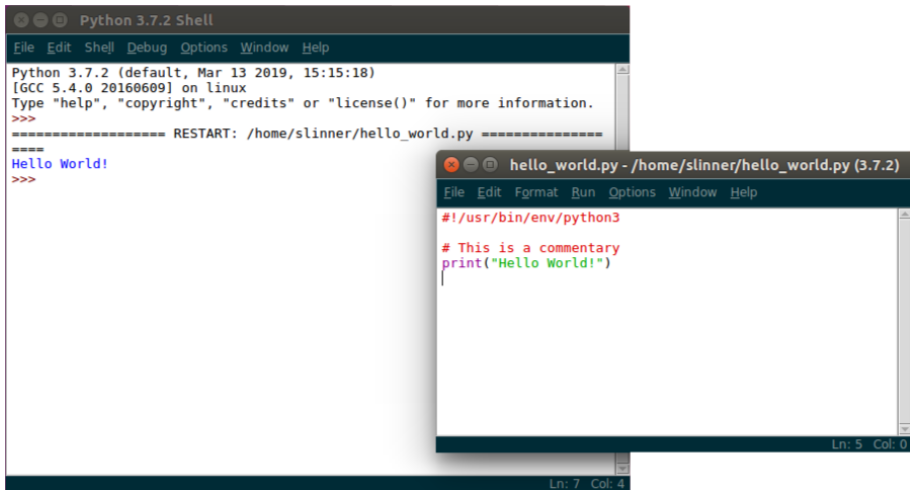
Interactive Mode

The interpreter can be started in interactive mode:

```
$ python3
Python 3.7.2 (default, Mar 13 2019, 15:15:18)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for
more information.
>>> print("hello world")
hello world
>>> a = 3 + 4
>>> print(a)
7
>>> 3 + 4
7
>>>
```

IDLE

- Integrated DeveLopment Environment
- Part of the Python installation



```
Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (default, Mar 13 2019, 15:15:18)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: /home/slinner/hello_world.py =====
Hello World!
>>>
```

```
hello_world.py - /home/slinner/hello_world.py (3.7.2)
File Edit Format Run Options Window Help
#!/usr/bin/env/python3
# This is a commentary
print("Hello World!")
Ln: 5 Col: 0
```

```
Ln: 7 Col: 4
```

Documentation

Online help in the interpreter:

- **help()**: general Python help
- **help(obj)**: help regarding an object, e.g. a function or a module
- **dir ()**: all used names
- **dir(obj)**: all attributes of an object

Official documentation: <http://docs.python.org/>

Documentation

```
>>> help(dir)
Help on built-in function dir:
...
>>> a = 3
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__', 'a']
>>> help(a)
Help on int object:
...
```


Differences Python 2 – Python 3 (incomplete)

	Python 2	Python 3
shebang ¹	<code>#!/usr/bin/python</code>	<code>#!/usr/bin/python3</code>
IDLE cmd ¹	<code>idle</code>	<code>idle3</code>
print cmd (syntax)	<code>print</code>	<code>print()</code>
input cmd (syntax)	<code>raw_input()</code>	<code>input()</code>
unicode	<code>u"..."</code>	all strings
integer type	<code>int/long</code>	<code>int</code> (infinite)
...	hints in each chapter	

⇒ <http://docs.python.org/3/whatsnew/3.0.html>

¹linux specific

Enjoy



Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

Numerical Data Types

- `int` : integer numbers (infinite)
- `float` : corresponds to `double` in C
- `complex` : complex numbers (`j` is the imaginary unit)

```
a = 1
```

```
c = 1.0
```

```
c = 1e0
```

```
d = 1 + 0j
```

Operators on Numbers

- **Basic arithmetics:** `+`, `-`, `*`, `/`

hint: *Python 2* \Rightarrow `1/2 = 0`

Python 3 \Rightarrow `1/2 = 0.5`

- **Div and modulo operator:** `//`, `%`, `divmod(x, y)`

- **Absolute value:** `abs(x)`

- **Rounding:** `round(x)`

- **Conversion:** `int(x)`, `float(x)`, `complex(re [, im=0])`

- **Conjugate of a complex number:** `x.conjugate()`

- **Power:** `x ** y`, `pow(x, y)`

Result of a composition of different data types is of the “bigger” data type.

Bitwise Operation on Integers

Operations:

- **AND:** `x & y`
- **OR:** `x | y`
- **exclusive OR (XOR) :**
`x ^ y`
- **invert:** `~x`
- **shift right n bits:** `x >> n`
- **shift left n bits:** `x << n`

Use `bin(x)` to get binary representation string of `x`.

```
>>> print(bin(6), bin(3))
0b110 0b11
>>> 6 & 3
2
>>> 6 | 3
7
>>> 6 ^ 3
5
>>> ~0
-1
>>> 1 << 3
8
>>> pow(2, 3)
8
>>> 9 >> 1
4
>>> print(bin(9), bin(9>>1))
0b1001 0b100
```

Strings

Data type: `str`

- `s = 'spam'`, `s = "spam"`
- Multiline strings: `s = """spam"""`
- No interpretation of escape sequences: `s = r"sp\nam"`
- Generate strings from other data types: `str(1.0)`

```
>>> s = """hello
... world"""
>>> print(s)
hello
world
>>> print("sp\nam")
sp
am
>>> print(r"sp\nam")    # or: print("sp\\nam")
sp\nam
```

String Methods

- Count appearance of substrings: `s.count(sub [, start[, end]])`
- Begins/ends with a substring? `s.startswith(sub[, start[, end]])`,
`s.endswith(sub[, start[, end]])`
- All capital/lowercase letters: `s.upper()`, `s.lower()`
- Remove whitespace: `s.strip([chars])`
- Split at substring: `s.split([sub [,maxsplit]])`
- Find position of substring: `s.index(sub[, start[, end]])`
- Replace a substring: `s.replace(old, new[, count])`

More methods: `help(str)`, `dir(str)`

Lists

Data type: `list`

- `s = [1, "spam", 9.0, 42]`, `s = []`
- **Append an element:** `s.append(x)`
- Extend with a second list: `s.extend(s2)`
- Count appearance of an element: `s.count(x)`
- Position of an element: `s.index(x[, min[, max]])`
- Insert element at position: `s.insert(i, x)`
- Remove and return element at position: `s.pop([i])`
- **Delete element:** `s.remove(x)`
- Reverse list: `s.reverse()`
- **Sort:** `s.sort([cmp[, key[, reverse]])`
- Sum of the elements: `sum(s)`

Tuple

Data type: `tuple`

- `s = 1, "spam", 9.0, 42`
`s = (1, "spam", 9.0, 42)`
- Constant list
- Count appearance of an element: `s.count(x)`
- Position of an element: `s.index(x[, min[, max]])`
- Sum of the elements: `sum(s)`

Tuple

Data type: `tuple`

- `s = 1, "spam", 9.0, 42`
`s = (1, "spam", 9.0, 42)`
- Constant list
- Count appearance of an element: `s.count(x)`
- Position of an element: `s.index(x[, min[, max]])`
- Sum of the elements: `sum(s)`

Multidimensional tuples and lists

- List and tuple can be nested (mixed):

```
>>> A = ([1, 2, 3], (1, 2, 3))
>>> A
([1, 2, 3], (1, 2, 3))
>>> A[0][2] = 99
>>> A
([1, 2, 99], (1, 2, 3))
```

Lists, Strings and Tuples

- Lists are **mutable**
- Strings and tuples are **immutable**
 - No assignment `s[i] = ...`
 - No appending and removing of elements
 - Functions like `x.upper()` return a new string!

```
>>> s1 = "spam"  
>>> s2 = s1.upper()  
>>> s1  
'spam'  
>>> s2  
'SPAM'
```

Operations on Sequences

Strings, lists and tuples have much in common: They are **sequences**.

- Does/doesn't s contain an element?

`x in s`, `x not in s`

- **Concatenate sequences:** `s + t`

- Multiply sequences: `n * s`, `s * n`

- **i-th element:** `s[i]`, i-th to last element: `s[-i]`

- Subsequence (slice): `s[i:j]`, with step size k: `s[i:j:k]`

- Subsequence (slice) from beginning/to end: `s[:-i]`, `s[i:]`, `s[:]`

- **Length** (number of elements): `len(s)`

- **Smallest/largest element:** `min(s)`, `max(s)`

- Assignments: `(a, b, c) = s`

→ `a = s[0]`, `b = s[1]`, `c = s[2]`

Indexing in Python

positive index	0	1	2	3	4	5	6	7	8	9	10
element	P	y	t	h	o	n		K	u	r	s
negative index	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> kurs = "Python Kurs"
>>> kurs[2:2]

>>> kurs[2:3]
t
>>> kurs[2]
t
>>> kurs[-4:-1]
Kur
>>> kurs[-4:]
Kurs
>>> kurs[-6:-8:-1]
no
```

Boolean Values

Data type **bool**: `True` , `False`

Values that are evaluated to `False` :

- `None` (data type `NoneType`)
- `False`
- `0` (in every numerical data type)
- Empty strings, lists and tuples: `''` , `[]` , `()`
- Empty dictionaries: `{}`
- Empty sets `set()`

All other objects of built-in data types are evaluated to `True` !

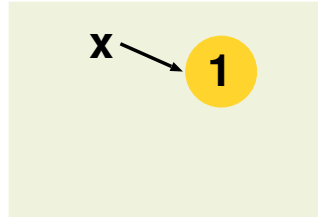
```
>>> bool([1, 2, 3])
True
>>> bool("")
False
```

References

- Every object name is a reference to this object!
- An assignment to a new name creates an additional reference to this object.
Hint: copy a list with `s2 = s1[:]` or `s2 = list(s1)`
- Operator `is` compares two references (identity),
operator `==` compares the contents of two objects
- Assignment: different behavior depending on object type
 - Strings, numbers (simple data types): create a new object with new value
 - Lists, dictionaries, ...: the original object will be changed

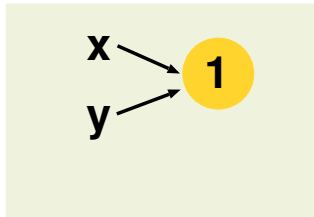
Reference - Example

```
>>> x=1
>>> y=x
>>> x is y
True
>>> y=2
>>> x is y
False
```



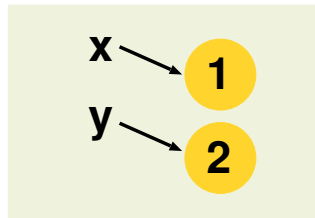
Reference - Example

```
>>> x=1
>>> y=x
>>> x is y
True
>>> y=2
>>> x is y
False
```



Reference - Example

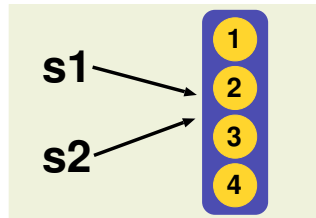
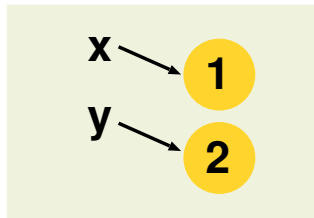
```
>>> x=1
>>> y=x
>>> x is y
True
>>> y=2
>>> x is y
False
```



Reference - Example

```
>>> x=1
>>> y=x
>>> x is y
True
>>> y=2
>>> x is y
False
```

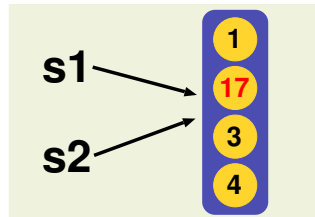
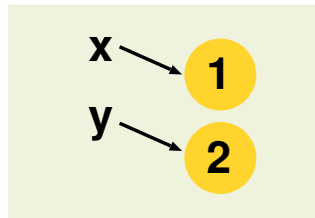
```
>>> s1 = [1, 2, 3, 4]
>>> s2 = s1
>>> s2[1] = 17
>>> s1
[1, 17, 3, 4]
>>> s2
[1, 17, 3, 4]
```



Reference - Example

```
>>> x=1
>>> y=x
>>> x is y
True
>>> y=2
>>> x is y
False
```

```
>>> s1 = [1, 2, 3, 4]
>>> s2 = s1
>>> s2[1] = 17
>>> s1
[1, 17, 3, 4]
>>> s2
[1, 17, 3, 4]
```



Groups

1	2	3	4
5	6	7	8

Enjoy



Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

The If Statement

```
if a == 3:  
    print("Aha!")
```

- Blocks are defined by indentation! \Rightarrow *Style Guide for Python*
- Standard: Indentation with four spaces

```
if a == 3:  
    print("spam")  
elif a == 10:  
    print("eggs")  
elif a == -3:  
    print("bacon")  
else:  
    print("something else")
```

Relational Operators

- Comparison of content: `==`, `<`, `>`, `<=`, `>=`, `!=`
- Comparison of object identity: `a is b`, `a is not b`
- And/or operator: `a and b`, `a or b`
- Chained comparison: `a <= x < b`, `a == b == c`, ...
- Negation: `not a`

```
if not (a==b) and (c<3):  
    pass
```

Hint: `pass` is a No Operation (NOOP) function

For Loops

```
for i in range(10):  
    print(i)    # 0, 1, 2, 3, ..., 9  
  
for i in range(3, 10):  
    print(i)    # 3, 4, 5, ..., 9  
  
for i in range(0, 10, 2):  
    print(i)    # 0, 2, 4, 6, 8  
else:  
    print("Loop completed.")
```

- End loop prematurely: `break`
- Next iteration: `continue`
- `else` is executed when loop didn't end prematurely

For Loops (continued)

Iterating directly over sequences (without using an index):

```
for item in ["spam", "eggs", "bacon"]:  
    print(item)
```

The `range` function can be used to create a list:

```
>>> list(range(0, 10, 2))  
[0, 2, 4, 6, 8]
```

If indexes are necessary:

```
for (i, char) in enumerate("hello world"):  
    print(i, char)
```

While Loops

```
i = 0
while i < 10:
    i += 1
```

`break` and `continue` work for while loops, too.

Substitute for do-while loop:

```
while True:
    # important code
    if condition:
        break
```

Enjoy



Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

Functions

```
def add(a, b):  
    """Returns the sum of a and b."""  
  
    mysum = a + b  
    return mysum
```

```
>>> result = add(3, 5)  
>>> print(result)  
8  
>>> help(add)  
Help on function add in module __main__:  
  
add(a, b)  
    Returns the sum of a and b.
```


Return Values and Parameters

- Functions accept arbitrary objects as parameters and return values
- Types of parameters and return values are unspecified
- Functions without explicit return value return `None`

my_program.py

```
def hello_world():  
    print("Hello World!")  
  
a = hello_world()  
print(a)
```

```
$ python3 my_program.py  
Hello World!  
None
```

Multiple Return Values

Multiple return values are realised using tuples or lists:

```
def foo():  
    a = 17  
    b = 42  
    return (a, b)  
  
ret = foo()  
(x, y) = foo()
```

Optional Parameters – Default Values

Parameters can be defined with default values.

Hint: It is not allowed to define non-default parameters after default parameters

plot_lines.py

```
def fline(x, m=1, b=0): #  $f(x) = m*x + b$ 
    return m*x + b

for i in range(5):
    print(fline(i), end=" ")
#force newline
print()
for i in range(5):
    print(fline(i, -1, 1), end=" ")
```

```
$ python3 plot_lines.py
0 1 2 3 4
1 0 -1 -2 -3
```

Hint: `end` in `print` defines the last character, default is linebreak

Positional Parameters

Parameters can be passed to a function in a different order than specified:

displayPerson.py

```
def printContact(name, age, location):  
    print("Person: ", name)  
    print("Age:      ", age, "years")  
    print("Address: ", location)  
  
printContact(name="Peter Pan", location="Neverland", age=10)
```

```
$ python3 displayPerson.py  
Person: Peter Pan  
Age:      10 years  
Address: Neverland
```

Functions are Objects

Functions are objects and as such can be assigned and passed on:

```
>>> a = float
>>> a(22)
22.0
```

```
>>> def foo(fkt):
...     print(fkt(33))
...
>>> foo(float)
33.0
>>> foo(str)
33
>>> foo(complex)
(33+0j)
```

Online Help: Docstrings

- Can be used in function, modul, class and method definitions
- Is defined by a **string** as the first statement in the definition
- `help(...)` on python object returns the docstring
- Two types of docstrings: **one-liners** and **multi-liners**

```
def complex(real=0.0, imag=0.0):  
    """Form a complex number.  
  
    Keyword arguments:  
    real -- the real part (default 0.0)  
    imag -- the imaginary part (default 0.0)  
  
    """  
    ...
```

Functions & Modules

- Functions thematically belonging together can be stored in a separate Python file. (Same for objects and classes)
- This file is called **module** and can be loaded in any Python script.
- Multiple modules available in the [Python Standard Library](#) (part of the Python installation)
- Command for loading a module: `import <filename>` (`filename` without ending `.py`)

```
import math
s = math.sin(math.pi)
```

More information for standard modules and how to create your own module see chapter [Modules and Packages](#) on slide 91

Enjoy



Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

String Formatting

- Format string + class method `x.format()`
- “replacement fields”: curly braces around optional `arg_name` (default: 0,1,2,...)

```
print("The answer is {0:4d}".format(42))
'The answer is    42'
s = "{0}: {1:08.3f}".format("spam", 3.14)
'spam: 0003.140'
```

format	purpose
	default: string
<code>m.nf</code>	floating point: m filed size, n digits after the decimal point (6)
<code>m.ne</code>	floating point (exponential): m filed size, 1 digit before and n digits behind the decimal point (default: 6)
<code>m.n%</code>	percentage: similar to format f , <i>value</i> * 100 with finalizing <code>'%'</code>
<code>md</code>	Integer number: m field size (0m \Rightarrow leading “0”) format d can be replaced by b (binary), o (octal) or x (hexadecimal)

Literal String Interpolation (f-strings)

- Provides a way to embed expressions inside string literals, using a minimal syntax
- Is a literal string, prefixed with 'f', which contains expressions inside braces
- Expressions are evaluated at runtime and replaced with their values.

```
>>> name = "Martin"
>>> age = 50
>>> f"My name is {name} and my age next year is {age+1}"
'My name is Martin and my age next year is 51'
>>> value = 12.345
>>> f"value={value:5.2f}"
'value=12.35'
```

Hint: Since Python 3.6!

String Formatting (deprecated, Python 2 only)

String formatting similar to C:

```
print "The answer is %4i." % 42
s = "%s: %08.3f" % ("spam", 3.14)
```

- **Integer decimal:** d, i
- **Integer octal:** o
- **Integer hexadecimal:** x, X
- **Float:** f, F
- **Float in exponential form:** e, E, g, G
- **Single character:** c
- **String:** s
- Use %% to output a single % character.

Command Line Input

User input in Python 3:

```
user_input = input("Type something: ")
```

User input in Python 2:

```
user_input = raw_input("Type something: ")
```

Hint: In Python 2 is `input("...")` \iff `eval(raw_input("..."))`

Command line parameters:

params.py

```
import sys
print(sys.argv)
```

```
$ python3 params.py spam
['params.py', 'spam']
```

Files

```
file1 = open("spam.txt", "r")  
file2 = open("/tmp/eggs.json", "wb")
```

- Read mode: `r`
- Write mode (new file): `w`
- Write mode, appending to the end: `a`
- Handling binary files: e.g. `rb`
- Read and write (update): `r+`

```
for line in file1:  
    print(line)
```

Operations on Files

- **Read:** `f.read([size])`
- **Read a line:** `f.readline()`
- **Read multiple lines:** `f.readlines([sizehint])`
- **Write:** `f.write(str)`
- **Write multiple lines:** `f.writelines(sequence)`
- **Close file:** `f.close()`

```
file1 = open("test.txt", "w")
lines = ["spam\n", "eggs\n", "ham\n"]
file1.writelines(lines)
file1.close()
```

Python automatically converts `\n` into the correct line ending!

The `with` statement

File handling (open/close) can be done by the context manager `with`.
(⇒section **Errors and Exceptions** on slide 65).

```
with open("test.txt") as f:  
    for line in f:  
        print(line)
```

After finishing the `with` block the file object is closed, even if an exception occurred inside the block.

Enjoy



Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

Syntax Errors, Indentation Errors

Parsing errors: **Program will not be executed.**

- Mismatched or missing parenthesis
- Missing or misplaced semicolons, colons, commas
- Indentation errors

add.py

```
print("I'm running...")
def add(a, b)
    return a + b
```

```
$ python3 add.py
File "add.py", line 2
    def add(a, b)
        ^
SyntaxError: invalid syntax
```

Exceptions

Exceptions occur at **runtime**:

error.py

```
import math
print("I'm running...")
math.foo()
print("I'm still running...")
```

```
$ python3 error.py
I'm running...
Traceback (most recent call last):
  File "error.py", line 3, in <module>
    math.foo()
AttributeError: module 'math' has no attribute 'foo'
```

Handling Exceptions (1)

```
try:  
    s = input("Enter a number: ")  
    number = float(s)  
except ValueError:  
    print("That's not a number!")
```

- `except` block is executed when the code in the `try` block throws an according exception
- Afterwards, the program continues normally
- Unhandled exceptions force the program to exit.

Handling different kinds of exceptions:

```
except (ValueError, TypeError, NameError):
```

Built-in exceptions: <http://docs.python.org/library/exceptions.html>

Handling Exceptions (2)

```
try:
    s = input("Enter a number: ")
    number = 1/float(s)
except ValueError:
    print("That's not a number!")
except ZeroDivisionError:
    print("You can't divide by zero!")
except:
    print("Oops, what's happened?")
```

- Several `except` statements for different exceptions
- Last `except` can be used without specifying the kind of exception: Catches all remaining exceptions
 - Careful: Can mask unintended programming errors!

Handling Exceptions (3)

- `else` is executed if no exception occurred
- `finally` is executed **in any** case

```
try:
    f = open("spam")
except IOError:
    print("Cannot open file")
else:
    print(f.read())
    f.close()
finally:
    print("End of try.")
```

Exception Objects

Access to exception objects:

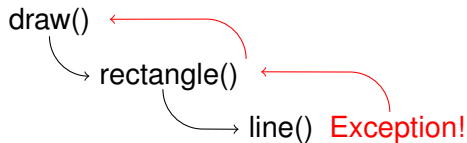
- `EnvironmentError` (`IOError`, `OSError`):
Exception object has 3 attributes (`errno`, `filename`, `strerror`)
- Otherwise: Exception object is a string

spam_open.py

```
try:  
    f = open("spam")  
except IOError as e:  
    print(e.errno, e.filename, e.strerror)  
    print(e)
```

```
$ python3 spam_open.py  
2 spam No such file or directory  
[Errno 2] No such file or directory: 'spam'
```


Exceptions in Function Calls



- Function calls another function.
- That function raises an exception.
- Is exception handled?
- No: Pass exception to calling function.

Raising Exceptions

Passing exceptions on:

```
try:
    f = open("spam")
except IOError:
    print("Problem while opening file!")
    raise
```

Raising exceptions:

```
def gauss_solver(matrix):
    # Important code
    raise ValueError("Singular matrix")
```

Exceptions vs. Checking Values Beforehand

Exceptions are preferable!

```
def square(x):  
    if type(x) == int or type(x) == float:  
        return x ** 2  
    else:  
        return None
```

- What about other numerical data types (complex numbers, own data types)? Better: Try to compute the power and catch possible exceptions! → **Duck-Typing**
- Caller of a function might forget to check return values for validity. Better: Raise an exception!

Exceptions vs. Checking Values Beforehand

Exceptions are preferable!

```
def square(x):  
    if type(x) == int or type(x) == float:  
        return x ** 2  
    else:  
        return None
```

```
def square(x):  
    return x ** 2  
...  
try:  
    result = square(value)  
except TypeError:  
    print("'{}': Invalid type".format(value))
```

The `with` Statement

Some objects offer context management ³, which provides a more convenient way to write `try ... finally` blocks:

```
with open("test.txt") as f:  
    for line in f:  
        print(line)
```

After the `with` block the file object is guaranteed to be closed properly, no matter what exceptions occurred within the block.

³Class method `__enter__(self)` will be executed at the beginning and class method `__exit__(...)` at the end of the context

Enjoy



Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

Sets

Set: unordered, no duplicated elements

- `s = {"a", "b", "c"}`

alternative `s = set([sequence])`, required for empty sets.

- **Constant set:** `s = frozenset([sequence])`

e.g. empty set: `empty = frozenset()`

- **Subset:** `s.issubset(t)`, `s <= t`, strict subset: `s < t`

- **Superset:** `s.issuperset(t)`, `s >= t`, strict superset: `s > t`

- **Union:** `s.union(t)`, `s | t`

- **Intersection:** `s.intersection(t)`, `s & t`

- **Difference:** `s.difference(t)`, `s - t`

- **Symmetric Difference:** `s.symmetric_difference(t)`, `s ^ t`

- **Copy:** `s.copy()`

As with sequences, the following works:

```
x in s, len(s), for x in s, s.add(x), s.remove(x)
```


Dictionaries

- Other names: Hash, Map, Associative Array
- Mapping of key → value
- Keys are unordered

```
>>> store = { "spam": 1, "eggs": 17}
>>> store["eggs"]
17
>>> store["bacon"] = 42
>>> store
{'eggs': 17, 'bacon': 42, 'spam': 1}
```

- Iterating over dictionaries:

```
for key in store:
    print(key, store[key])
```

- Compare two dictionaries: `store == pool`

Not allowed: `>`, `>=`, `<`, `<=`

Operations on Dictionaries

- Delete an entry: `del(store[key])`
- Delete all entries: `store.clear()`
- Copy: `store.copy()`
- Does it contain a key? `key in store`
- Get an entry: `store.get(key[, default])`
- Remove and return entry: `store.pop(key[, default])`
- Remove and return arbitrary entry: `store.popitem()`

Operations on Dictionaries

- Delete an entry: `del(store[key])`
- Delete all entries: `store.clear()`
- Copy: `store.copy()`
- Does it contain a key? `key in store`
- Get an entry: `store.get(key[, default])`
- Remove and return entry: `store.pop(key[, default])`
- Remove and return arbitrary entry: `store.popitem()`

Views on Dictionaries

- Create a view: `items()`, `keys()` and `values()`
 - List of all (key, value) tuples: `store.items()`
 - List of all keys: `store.keys()`
 - List all values: `store.values()`
- **Caution:** Dynamical since Python 3

Views Behavior: Python 2.X versus Python 3.X

Python 2 (static)

```
>>> mdict={"a":2, "d":5}
>>> mdict
{'a': 2, 'd': 5}
>>> s=mdict.items()
>>> for i in s:
    print(i)
('a', 2)
('d', 5)
>>> mdict['a']=-1
>>> mdict
{'a': -1, 'd': 5}
>>> for i in s:
    print(i)
('a', 2)
('d', 5)
```

Python 3 (dynamic)

```
>>> mdict={"a":2, "d":5}
>>> mdict
{'a': 2, 'd': 5}
>>> s=mdict.items()
>>> for i in s:
    print(i)
('a', 2)
('d', 5)
>>> mdict['a']=-1
>>> mdict
{'a': -1, 'd': 5}
>>> for i in s:
    print(i)
('a', -1)
('d', 5)
```

Enjoy



Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

Object Oriented Programming (OOP)

- So far: **procedural programming**
 - Data (values, variables, parameters, ...)
 - Functions taking data as parameters and returning results
- Alternative: Group data and functions belonging together to form **custom data types**
- → Extensions of structures in C/Fortran

Using Simple Classes as Structs

my_point.py

```
class Point:
    pass

p = Point()
p.x = 2.0
p.y = 3.3
```

- **Class:** Custom data type (here: `Point`)
- **Object:** Instance of a class (here: `p`)
- **Attributes** (here `x`, `y`) can be added dynamically

Hint: `pass` is a No Operation (NOOP) function

Classes - Constructor

my_point.py

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Point(2.0, 3.0)
print(p.x, p.y)
p.x = 2.5
p.z = 42
```

- `__init__` : Is called automatically after creating an object

Methods on Objects

my_point.py

```
import math

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

    def norm(self):
        n = math.sqrt(self.x**2 + self.y**2)
        return n

p = Point(2.0, 3.0)
print(p.x, p.y, p.norm())
```

- Method call: automatically sets the object as first parameter
- → traditionally called `self`
- **Careful**: Overloading of methods not possible!

Converting Objects to Strings

Default return value of `str(...)` for objects of custom classes:

```
>>> p = Point(2.0, 3.0)
>>> print(p) # --> print(str(p))
<__main__.Point instance at 0x402d7a8c>
```

Converting Objects to Strings

Default return value of `str(...)` for objects of custom classes:

```
>>> p = Point(2.0, 3.0)
>>> print(p) # --> print(str(p))
<__main__.Point instance at 0x402d7a8c>
```

This behaviour can be overwritten:

my_point.py

```
class Point:
    [...]
    def __str__(self):
        return "{0}, {1}".format(self.x, self.y)
```

```
>>> print(p)
(2.0, 3.0)
```

Comparing Objects

Default: `==` checks for object identity of custom objects.

```
>>> p1 = Point(2.0, 3.0)
>>> p2 = Point(2.0, 3.0)
>>> p1 == p2
False
```

Comparing Objects

Default: `==` checks for object identity of custom objects.

```
>>> p1 = Point(2.0, 3.0)
>>> p2 = Point(2.0, 3.0)
>>> p1 == p2
False
```

This behaviour can be overwritten:

my_point.py

```
class Point:
    [...]
    def __eq__(self, other):
        return (self.x == other.x) and (self.y == other.y)
```

```
>>> p1 == p2 # Check for equal values
True
>>> p1 is p2 # Check for identity
False
```

Operator overloading

More relational operators:

- `<` : `__lt__(self, other)`
- `<=` : `__le__(self, other)`
- `!=` : `__ne__(self, other)`
- `>` : `__gt__(self, other)`
- `>=` : `__ge__(self, other)`

Numeric operators:

- `+` : `__add__(self, other)`
- `-` : `__sub__(self, other)`
- `*` : `__mul__(self, other)`
- ...

Emulating Existing Data Types

Classes can emulate built-in data types:

- Numbers: arithmetics, `int(myobj)`, `float(myobj)`, ...
- Functions: `myobj(...)`
- Sequences: `len(myobj)`, `myobj[...]`, `x in myobj`, ...
- Iterators: `for i in myobj`

See documentation: <http://docs.python.org/3/reference/datamodel.html>

Class Variables

Have the same value for all instances of a class:

my_point.py

```
class Point:
    count = 0 # Count all point objects
    def __init__(self, x, y):
        Point.count += 1 #self.__class__.count += 1
    [...]
```

```
>>> p1 = Point(2, 3); p2 = Point(3, 4)
>>> p1.count
2
>>> p2.count
2
>>> Point.count
2
```

Class Methods and Static Methods

spam.py

```
class Spam:
    spam = "I don't like spam."

    @classmethod
    def cmethod(cls):
        print(cls.spam)

    @staticmethod
    def smethod():
        print("Blah blah.")
```

```
Spam.cmethod()
Spam.smethod()
s = Spam()
s.cmethod()
s.smethod()
```

Inheritance (1)

There are often classes that are very similar to each other.

Inheritance allows for:

- Hierarchical class structure (is-a-relationship)
- Reusing of similar code

Example: Different types of phones

- Phone
- Mobile phone (is a phone with additional functionality)
- Smart phone (is a mobile phone with additional functionality)

Inheritance (2)

```
class Phone:
    def call(self):
        pass

class MobilePhone(Phone):
    def send_text(self):
        pass
```

MobilePhone now inherits methods and attributes from Phone.

```
h = MobilePhone()
h.call() # inherited from Phone
h.send_text() # own method
```

Overwriting Methods

Methods of the parent class can be overwritten in the child class:

```
class MobilePhone(Phone):  
    def call(self):  
        self.find_signal()  
        Phone.call(self)
```

Multiple Inheritance

Classes can inherit from multiple parent classes. Example:

- SmartPhone is a mobile phone
- SmartPhone is a camera

```
class SmartPhone(MobilePhone, Camera):  
    pass  
  
h = SmartPhone()  
h.call() # inherited from MobilePhone  
h.take_photo() # inherited from Camera
```

Attributes are searched for in the following order:

SmartPhone , MobilePhone , parent class of MobilePhone (recursively), Camera ,
parent class of Camera (recursively).

Private Attributes / Private Class Variables

- There are no private variables or private methods in Python.
- **Convention:** Mark attributes that shouldn't be accessed from outside with an underscore: `_foo`.
- To avoid name conflicts during inheritance: Names of the form `__foo` are replaced with `_classname__foo`:

```
class Spam:  
    __eggs = 3  
    _bacon = 1  
    beans = 5
```

```
>>> dir(Spam)  
>>> ['_Spam__eggs', '__doc__', '__module__', '_bacon', 'beans']
```

Classic (old Style) Classes

- The only class type until Python 2.1
- In Python 2 default class

New Style Classes

- Unified class model (user-defined and build-in)
- Descriptors (getter, setter)
- The only class type in Python 3
- Available as basic class in Python 2: **object**

Properties (1)

If certain actions (checks, conversions) are to be executed while accessing attributes, use **getter** and **setter**:

```
class Spam:
    def __init__(self):
        self._value = 0

    def get_value(self):
        return self._value

    def set_value(self, value):
        if value <= 0:
            self._value = 0
        else:
            self._value = value

    value = property(get_value, set_value)
```

Properties (2)

Properties can be accessed like any other attributes:

```
>>> s = Spam()
>>> s.value = 6      # set_value(6)
>>> s.value         # get_value()
6
>>> s.value = -6   # set_value(-6)
>>> s.value         # get_value()
0
```

- Getter and setter can be added later without changing the API
- Access to `_value` still possible

Enjoy



Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

Importing Modules

Reminder: Functions, classes and object thematically belonging together are grouped in modules.

```
import math  
s = math.sin(math.pi)
```

```
import math as m  
s = m.sin(m.pi)
```

```
from math import pi as PI, sin  
s = sin(PI)
```

```
from math import *  
s = sin(pi)
```

Online help: `dir(math)` , `help(math)`

Creating a Module (1)

Every Python script can be imported as a module.

my_module.py

```
"""My first module: my_module.py"""  
  
def add(a, b):  
    """Add a and b."""  
    return a + b  
  
print(add(2, 3))
```

```
>>> import my_module  
5  
>>> my_module.add(17, 42)  
59
```

Top level instructions are executed during import!

Creating a Module (2)

If instructions should only be executed when running as a script, not importing it:

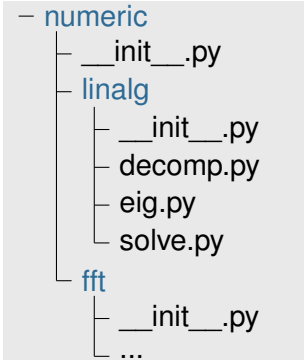
my_module.py

```
def add(a, b):  
    return a + b  
  
def main():  
    print(add(2, 3))  
  
if __name__ == "__main__":  
    main()
```

Useful e.g. for testing parts of the module.

Creating a Package

Modules can be grouped into hierarchically structured packages.



- Packages are subdirectories
- In each package directory:
 - `__init__.py` (may be empty)

```
import numeric
numeric.foo() # from __init__.py
numeric.linalg.eig.foo()
```

```
from numeric.linalg import eig
eig.foo()
```


Modules Search Path

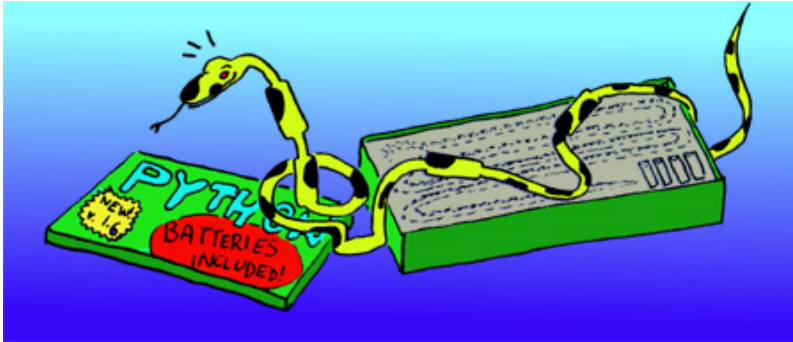
Modules are searched for in (see `sys.path`):

- The directory of the running script
- Directories in the environment variable PYTHONPATH
- Installation-dependent directories

```
>>> import sys
>>> sys.path
['', '/usr/lib/python37.zip',
'/usr/lib64/python3.7',
'/usr/lib64/python3.7/plat-linux', ...]
```

Python's Standard Library

„**Batteries included**“: comprehensive standard library for various tasks



Mathematics: `math`

- Constants: `e`, `pi`
- Round up/down: `floor(x)`, `ceil(x)`
- Exponential function: `exp(x)`
- Logarithm: `log(x[, base])`, `log10(x)`
- Power and square root: `pow(x, y)`, `sqrt(x)`
- Trigonometric functions: `sin(x)`, `cos(x)`, `tan(x)`
- Conversion degree \leftrightarrow radian: `degrees(x)`, `radians(x)`

```
>>> import math
>>> math.sin(math.pi)
1.2246063538223773e-16
>>> math.cos(math.radians(30))
0.86602540378443871
```

Random Numbers: `random`

- Random integers:

```
randint(a, b), randrange([start,] stop[, step])
```

- Random floats (uniform distr.): `random()`, `uniform(a, b)`

- Other distributions: `expovariate(lambd)`, `gammavariate(alpha, beta)`, `gauss(mu, sigma)`, ...

- Random element of a sequence: `choice(seq)`

- Several unique, random elements of a sequence: `sample(population, k)`

- Shuffled sequence: `shuffle(seq[, random])`

```
>>> import random
>>> s = [1, 2, 3, 4, 5]
>>> random.shuffle(s)
>>> s
[2, 5, 4, 3, 1]
>>> random.choice("Hello world!")
'e'
```

Time Access and Conversion: `time`

- Classical `time()` functionality
- Time class type is a 9-tuple of `int` values (`struct_time`)
- Time starts at `epoch` (for UNIX: 1.1.1970, 00:00:00)
- Popular functions:
 - Seconds since `epoch` (as a float): `time.time()`
 - Convert time in seconds (float) to `struct_time`: `time.localtime([seconds])`
If seconds is `None` the actual time is returned.
 - Convert `struct_time` in seconds (float): `time.mktime(t)`
 - Convert `struct_time` in formatted string: `time.strftime(format[, t])`
 - Suspend execution of current thread for `secs` seconds: `time.sleep(secs)`

Date and Time: `datetime`

Date and time objects:

```
d1 = datetime.date(2008, 3, 21)
d2 = datetime.date(2008, 6, 22)
dt = datetime.datetime(2011, 8, 26, 12, 30)
t = datetime.time(12, 30)
```

Calculating with date and time:

```
print(d1 < d2)
delta = d2 - d1
print(delta.days)
print(d2 + datetime.timedelta(days=44))
```

Operations on Path Names: `os.path`

- Paths: `abspath(path)`, `basename(path)`, `normpath(path)`, `realpath(path)`
- Construct paths: `join(path1[, path2[, ...]])`
- Split paths: `split(path)`, `splittext(path)`
- File information: `isfile(path)`, `isdir(path)`, `islink(path)`, `getsize(path)`,
...
- Expand home directory: `expanduser(path)`
- Expand environment variables: `expandvars(path)`

```
>>> os.path.join("spam", "eggs", "ham.txt")
'spam/eggs/ham.txt'
>>> os.path.splittext("spam/eggs.py")
('spam/eggs', '.py')
>>> os.path.expanduser("~/spam")
'/home/rbreu/spam'
>>> os.path.expandvars("/mydir/$TEST")
'/mydir/test.py'
```

Files and Directories: `os`

- Working directory: `getcwd()`, `chdir(path)`
- Changing file permissions: `chmod(path, mode)`
- Changing owner: `chown(path, uid, gid)`
- Creating directories: `makedirs(path[, mode])`, `makedirs(path[, mode])`
- Removing files: `remove(path)`, `rmdir(path)`
- Renaming files: `rename(src, dst)`, `rename(old, new)`
- List of files in a directory: `listdir(path)`

```
for myfile in os.listdir("mydir"):  
    os.chmod(os.path.join("mydir", myfile),  
            os.path.stat.S_IRGRP)
```


Files and Directories: `shutil`

Higher level operations on files and directories. Mighty wrapper functions for `os` module.

- Copying files: `copyfile(src, dst)`, `copy(src, dst)`
- Recursive copy: `copytree(src, dst[, symlinks])`
- Recursive removal:
`rmtree(path[, ignore_errors[, onerror]])`
- Recursive move: `move(src, dst)`

```
shutil.copytree("spam/eggs", "../beans",  
               symlinks=True)
```

Directory Listing: `glob`

List of files in a directory with Unix-like extension of wildcards: `glob(path)`

```
>>> glob.glob("python/[a-c]*.py")
['python/confitest.py',
 'python/basics.py',
 'python/curses_test2.py',
 'python/curses_keys.py',
 'python/cmp.py',
 'python/button_test.py',
 'python/argument.py',
 'python/curses_test.py']
```

Run Processes: subprocess

Simple execution of a program:

```
p = subprocess.Popen(["ls", "-l", "mydir"])  
returncode = p.wait() # wait for p to end
```

Access to the program's output:

```
p = Popen(["ls"], stdout=PIPE, stderr=STDOUT)  
p.wait()  
output = p.stdout.read()
```

Pipes between processes (`ls -l | grep txt`)

```
p1 = Popen(["ls", "-l"], stdout=PIPE)  
p2 = Popen(["grep", "txt"], stdin=p1.stdout)
```

Access to Command Line Parameters: `argparse` (1)

Python program with standard command line option handling:

```
$ ./argumentParser.py -h
usage: argumentParse.py [-h] -f FILENAME [-v]
```

Example how to use argparse

optional arguments:

<code>-h, --help</code>	show this help message and exit
<code>-f FILENAME, --file FILENAME</code>	output file
<code>-v, --verbosity</code>	increase output verbosity

```
$ python3 argumentParse.py -f newfile.txt -v
newfile.txt
True
```

Access to Command Line Parameters: `argparse` (2)

- Simple list of parameters: → `sys.argv`
- More convenient for handling several options: `argparse`
- Deprecated module `optparse` (since Python 2.7/3.2)

argumentParse.py

```
parser = argparse.ArgumentParser(  
    description='Example how to use argparse')  
parser.add_argument("-f", "--file",  
                    dest="filename",  
                    default="out.txt",  
                    help="output file")  
parser.add_argument("-v", "--verbosity",  
                    action="store_true",  
                    help="increase output verbosity")  
  
args = parser.parse_args()  
print(args.filename)  
print(args.verbosity)
```

CSV Files: CSV (1)

CSV: Comma Separated Values

- Data tables in ASCII format
- Import/Export by MS Excel[®]
- Columns are delimited by a predefined character (most often comma)

```
f = open("test.csv", "r")
reader = csv.reader(f)
for row in reader:
    for item in row:
        print(item)
f.close()
```

```
f = open(outfile, "w")
writer = csv.writer(f)
writer.writerow([1, 2, 3, 4])
```

CSV Files: CSV (2)

Handling different kinds of formats (dialects):

```
csv.reader(csvfile, dialect='excel') # Default  
csv.writer(csvfile, dialect='excel_tab')
```

Specifying individual format parameters:

```
csv.reader(csvfile, delimiter=";")
```

Further format parameters: `lineterminator`, `quotechar`, `skipinitialspace`, ...

Lightweight Database: `sqlite3` (1)

Database in a file or in memory; in Python's stdlib since 2.5.

```
conn = sqlite3.connect("bla.db")
c = conn.cursor()

c.execute("""CREATE TABLE Friends
            (firstname TEXT, lastname TEXT)""")
c.execute("""INSERT INTO Friends
            VALUES ("Jane", "Doe")""")
conn.commit()
```

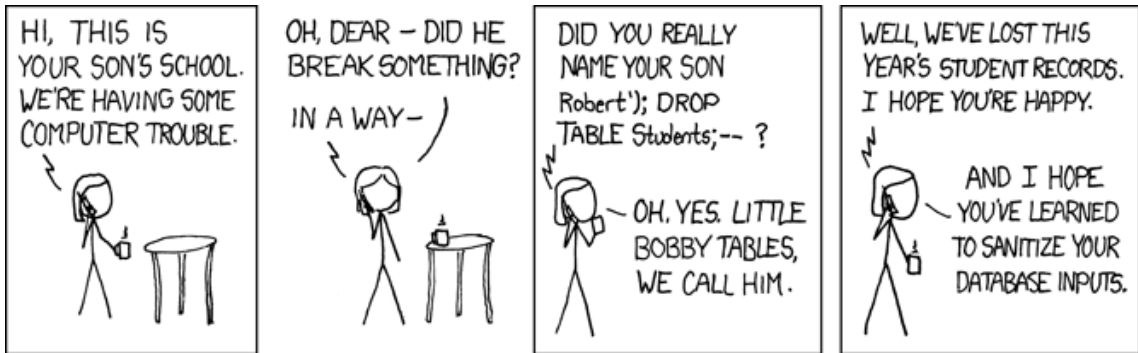
```
c.execute("""SELECT * FROM Friends""")
for row in c:
    print(row)

c.close();
conn.close()
```


Lightweight Database: `sqlite3` (2)

String formatting is insecure since it allows injection of arbitrary SQL code!

```
# Never do this!  
symbol = "Jane"  
c.execute("... WHERE firstname='{0}'".format(symbol))
```



Lightweight Database: `sqlite3` (3)

Instead: Use the placeholder the database API provides:

```
c.execute("... WHERE name = ?", symbol)
```

```
friends = (("Janis", "Joplin"), ("Bob", "Dylan"))
for item in friends:
    c.execute("""INSERT INTO Friends
                VALUES (?,?)""", item)
```

⇒ Python module `cx_Oracle` to access Oracle database
Web page: <http://cx-oracle.sourceforge.net/>

XML based Client-Server Communication: `xmlrpc` (1)

- XML-RPC: **Remote Procedure Call** uses XML via HTTP
- Independent of platform and programming language
- For the client use `xmlrpc.client`

```
import xmlrpc.client

s = xmlrpc.client.Server("http://localhost:8000")
# print list of available methods
print(s.system.listMethods())
# use methods
print(s.add(2,3))
print(s.sub(5,2))
```

Automatic type conversion for the standard data types: boolean, integer, floats, strings, tuple, list, dictionaries (strings as keys), ...

XML based Client-Server Communication: `xmlrpc` (2)

- For the server use `xmlrpc.server`

```
from xmlrpc.server import SimpleXMLRPCServer

# methods which are to be offered by the server:
class MyFuncs:
    def add(self, x, y):
        return x + y
    def sub(self, x, y):
        return x - y

# create and start the server:
server = SimpleXMLRPCServer(("localhost", 8000))
server.register_instance(MyFuncs())
server.serve_forever()
```

More Modules

- `readline` : Functionality for command line history and auto-completion
- `tempfile` : Generate temporary files and directories
- `numpy` : **N**umeric **P**ython package
 - N-dimensional arrays
 - Supports linear algebra, Fourier transform and random number capabilities
 - Part of the `SciPy` stack
- `matplotlib` : 2D plotting library, part of the `SciPy` stack
- ...

Enjoy



Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

Conditional Expressions

A conditional assignment as

```
if value < 0:  
    s = "negative"  
else:  
    s = "positive"
```

can be realized in abbreviated form

```
s = "negative" if value < 0 else "positive"
```


List Comprehension

Allows sequences to be build by sequences. Instead of using `for` :

```
a = []  
for i in range(10):  
    a.append(i**2)
```

List comprehension can be used:

```
a = [i**2 for i in range(10)]
```

Conditional values in list comprehension:

```
a = [i**2 for i in range(10) if i != 4]
```

Since **Python 2.7**: set and dictionary comprehension

```
s = {i*2 for i in range(3)}  
d = {i: i*2 for i in range(3)}
```

Dynamic Attributes

Remember: Attributes can be added to python objects at runtime:

```
class Empty:  
    pass  
  
a = Empty()  
a.spam = 42  
a.eggs = 17
```

Also the attributes can be deleted at runtime:

```
del(a.spam)
```

getattr, setattr, hasattr

Attributes of an object can be accessed by name (string):

```
import math
f = getattr(math, "sin")
print(f(x)) # sin(x)
```

```
a = Empty()
setattr(a, "spam", 42)
print(a.spam)
```

Useful if depending on user or data input.

Check if attribute is defined:

```
if not hasattr(a, "spam"):
    setattr(a, "spam", 42)
print(a.spam)
```

Anonymous Function Lambda

Also known as `lambda expression` and `lambda form`

```
>>> f = lambda x, y: x + y
>>> f(2, 3)
5
>>> (lambda x: x**2)(3)
9
```

Useful if only a simple function is required as an parameter in a function call:

```
>>> friends = ["alice", "Bob"]
>>> friends.sort()
>>> friends
['Bob', 'alice']
>>> friends.sort(key = lambda a: a.upper())
>>> friends
['alice', 'Bob']
```

Functions Parameters from Lists and Dictionaries

```
def spam(a, b, c, d):  
    print(a, b, c, d)
```

Positional parameters can be created by lists:

```
>>> args = [3, 6, 2, 3]  
>>> spam(*args)  
3 6 2 3
```

Keyword parameters can be created by dictionaries:

```
>>> kwargs = {"c": 5, "a": 2, "b": 4, "d":1}  
>>> spam(**kwargs)  
2 4 5 1
```

Variable Number of Parameters in Functions

```
def spam(*args, **kwargs):  
    for i in args:  
        print(i)  
    for i in kwargs:  
        print(i, kwargs[i])
```

```
>>> spam(1, 2, c=3, d=4)  
1  
2  
c 3  
d 4
```

Global and Static Variables in Functions

- `global` links the given name to a global variable
- Static variable can be defined as an attribute of the function

```
def myfunc():
    global max_size
    if not hasattr(myfunc, "_counter"):
        myfunc._counter = 0 # it doesn't exist yet,
                            # so initialize it

    myfunc._counter += 1
    print("{0:d}. call".format(myfunc._counter))
    print("max size is {0:d}".format(max_size))
...

```

```
>>> max_size = 222
>>> myfunc()
1. call
max size is 222

```

Map

Apply specific function on each list element:

```
>>> li = [1, 4, 81, 9]
>>> mapli = map(math.sqrt, li)
>>> mapli
<map object at 0x7f5748240b90>
>>> list(mapli)
[1.0, 2.0, 9.0, 3.0]
>>> list(map(lambda x: x * 2, li))
[2, 8, 162, 18]
```

Functions with more than one parameter requires an additional list per parameter:

```
>>> list(map(math.pow, li, [1, 2, 3, 4]))
[1.0, 16.0, 531441.0, 6561.0]
```


Filter

Similar to `map`, but the result is a `filter` object, which contains only list elements, where the function returns `True`.

filter_example.py

```
li = [1, 2, 3, 4, 5, 6, 7, 8, 9]
liOdd = filter(lambda x: x % 2, li)
print("li =", li)
print("liOdd =", liOdd)
print("list(liOdd) =", list(liOdd))
```

```
$ python3 filter_example.py
li = [1, 2, 3, 4, 5, 6, 7, 8, 9]
liOdd = <filter object at 0x7fe4ccdc7c0>
list(liOdd) = [1, 3, 5, 7, 9]
$
```

Zip

- Join multiple sequences to one list of tuples:
Useful when iterating on multiple sequences in parallel

```
>>> list(zip("ABC", "123"))  
[('A', '1'), ('B', '2'), ('C', '3')]  
>>> list(zip([1, 2, 3], "ABC", "XYZ"))  
[(1, 'A', 'X'), (2, 'B', 'Y'), (3, 'C', 'Z')]
```

- Example: How to create a dictionary by two sequences

```
>>> dict(zip(("apple", "peach"), (2,0)))  
{'apple': 2, 'peach': 0}
```

Iterators (1)

What happens, if `for` is applied on an object?

```
for i in obj:  
    pass
```

- The `__iter__` method for `obj` is called, return an **iterator**.
- On each loop cycle the `iterator.__next__()` method will be called.
- The exception `StopIteration` is raised when there are no more elements.
- Advantage: Memory efficient (access time)

Iterators (2)

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            self.index = len(self.data)
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> for char in Reverse("spam"):
...     print(char, end=" ")
...
m a p s
```

Generators

Simple way to create iterators:

- Methods uses the `yield` statement
⇒ breaks at this point, returns element and continues there on the next `iterator.__next__()` call.

```
def reverse(data):  
    for element in data[::-1]:  
        yield element
```

```
>>> for char in reverse("spam"):  
...     print(char, end=" ")  
...  
m a p s
```

Generator Expressions

Similar to the **list comprehension** an **iterator** can be created using a **generator expression**:

```
>>> data = "spam"
>>> for c in (elem for elem in data[::-1]):
...     print(c, end=" ")
...
m a p s
```

Enjoy



Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

IPython (I)

- Enhanced interactive Python shell
- Numbered input/output prompts
- Object introspection

```
In [1]: len?  
Type:      builtin_function_or_method  
String Form: <built-in function len>  
Namespace: Python builtin  
Docstring:  
len(object)  
  
Return the number of items of a sequence or collection.
```

- System shell access

```
In [1]: a = !ls  
  
In [2]: print(a)  
['example01.py', 'example02.py', 'example03.py']
```

IPython (II)

- Tab-completion
- Command history retrieval across session
- User-extensible 'magic' commands
 - `%timeit` ⇒ Time execution of a Python statement or expression using the `timeit` module
 - `%cd` ⇒ Change the current working directory
 - `%edit` ⇒ Bring up an editor and execute the resulting code
 - `%run` ⇒ Run the named file inside IPython as a program
 - ⇒ *more 'magic' commands*
- ⇒ *IPython documentation*

PIP Installs Python/Packages (I)

- Command `pip`
- A tool for installing Python packages
- Python 2.7.9 and later (on the python2 series), and Python 3.4 and later include `pip` by default
- Installing Packages

```
$ pip3 install SomePackage  
$ pip3 install --user SomePackage #user install
```

- Uninstall Packages

```
$ pip3 uninstall SomePackage
```

PIP Installs Python/Packages (II)

- Listing Packages

```
$ pip3 list
docutils (0.9.1)
Jinja2 (2.10)
Pygments (2.3.1)
Sphinx (1.1.2)
$ pip3 list --outdated
docutils (Current: 0.9.1 Latest: 0.14)
Sphinx (Current: 1.1.2 Latest: 2.10)
```

- Searching for Packages

```
$ pip3 search "query"
```

- ⇒ *pip documentation*

pyenv - Simple Python Version Management (I)

- Easily switch between multiple versions of Python
- Doesn't depend on Python itself
- Inserts directory of *shims*⁴ at the front of your `PATH`
- Easy Installation:

```
$ git clone https://github.com/yyuu/pyenv.git ~/.pyenv
$ echo 'export PYENV_ROOT="$ HOME/.pyenv"' >> ~/.bashrc
$ echo 'export PATH="$ PYENV_ROOT/bin:$ PATH"' >> ~/.bashrc
$ echo 'eval "$ (pyenv init -)"' >> ~/.bashrc
```

- ⇒ *pyenv repository*

⁴kind of infrastructure to redirect system/function calls
metaphor: A *shim* is a piece of wood or metal to make two things fit together

pyenv - Simple Python Version Management (II)

- Install Python versions into `$PYENV_ROOT/versions`

```
$ pyenv install --list      # available Python versions
$ pyenv install 3.7.4      # install Python 3.7.4
```

- Change the Python version

```
$ pyenv global 3.7.4      # global Python
$ pyenv local 3.7.4      # per-project Python
$ pyenv shell 3.7.4      # shell-specific Python
```

- List all installed Python versions (asterisk shows the active)

```
$ pyenv versions
system
2.7.16
* 3.7.4 (set by PYENV_VERSION environment variable)
```

Virtual Environments

- Allow Python packages to be installed in an isolated location
- Use cases
 - Two applications need different versions of a library
 - Install an application and leave it be
 - Can't install packages into the global site-packages directory
- Virtual environments have their own installation directories
- Virtual environments don't share libraries with other virtual environments
- Available implementations:
 - `virtualenv` (Python 2 and Python 3)
 - `venv` (Python 3.3 and later)

venv

- Create virtual environment

```
$ python3 -m venv /path/to/env
```

- Activate

```
$ source /path/to/env/bin/activate
```

- Deactivate

```
$ deactivate
```

- ⇒ *venv documentation*

Pylint (I)

- `pylint` is the `lint` implementation for python code
- Checks for errors in Python code
- Tries to enforce a coding standard
- Looks for bad code smells
- Displays classified messages under various categories such as errors and warnings
- Displays statistics about the number of warnings and errors found in different files

Pylint (II)

- The code is given an overall mark

```
$ python3 -m pylint example.py

...

Global evaluation
-----
Your code has been rated at 10.00/10
      (previous run: 9.47/10, +0.53)
```

- ⇒ *Pylint documentation*

Software testing

- Part of quality management
- Point out the defects and errors that were made during the development phases
- It always ensures the users or customers satisfaction and reliability of the application
- The cost of fixing the bug is larger if testing is not done \Rightarrow testing saves time
- Python testing tools
 - pytest
 - unittest
 - ...

pytest

- Easy to get started
- `test_` prefixed test functions or methods are test items
- Asserting with the `assert` statement
- pytest will run all files in the current directory and its subdirectories of the form `test_*.py` or `*_test.py`
- Usage:

```
$ python3 -m pytest
...
$ python3 -m pytest example.py
...
```

- ⇒ *pytest documentation*

pytest Example: Check Function Return Value

example1_test.py

```
def incr(x):  
    return x + 11  
  
def test_incr():  
    assert incr(3) == 4
```

```
$ python3 -m pytest -v example1_test.py  
...  
----- test_incr -----  
def test_incr():  
>     assert incr(3) == 4  
E     assert 14 == 4  
E         + where 14 = incr(3)  
  
example1_test.py:5: AssertionError  
===== 1 failed in 0.00 seconds =====
```

pytest Example: Check for expected Exception

```
import pytest

def f():
    raise SystemExit(1)

def test_error():
    with pytest.raises(SystemExit): #passes
        f()
```

pytest Example: Check for expected Exception

```
import pytest

def f():
    raise SystemExit(1)

def test_error():
    with pytest.raises(SystemExit): #passes
        f()
```

pytest Example: Comparing Two Data Object

```
def test_list_comparison():
    list1 = [1,3,0,8]
    list2 = [1,3,3,8]
    assert list1 == list2 #fails
```

pytest Example: Parameterize Test Function

```
def incr(x):  
    return x + 1  
  
@pytest.mark.parametrize("test_input, expected", [  
    (1, 2),  
    (2, 3),  
    (3, 4),  
])  
def test_incr(test_input, expected):  
    assert incr(test_input) == expected
```


Enjoy



Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

Regular Expressions – Introduction

- Regular expression (RegExp):
Formal language for pattern matching in strings
- Motivation: Analyze various text files:
 - Log files
 - Data files (e.g. experimental data, system configuration, ...)
 - Command output
 - ...
- Python module: `import re`

```
>>> re.findall(r"a.c", "abc aac aa abb a c")  
['abc', 'aac', 'a c']
```

Remember:

`r"..."` ⇒ raw string (escape sequences are not interpreted)

Regular Expressions – Character Classes

- Class/set of possible characters: `[!?:.,;a-z]`
- `^` at the beginning negates the class.
e.g.: `[^aeiou]` ⇒ all characters besides the vocals
- Character class in pattern tests for **one** character
- The `.` represents **any** (one) character
- Predefined character classes:

name	character	Acr.	negated
whitespace	<code>[\t\n\r\f]</code>	<code>\s</code>	<code>\S</code>
word character	<code>[a-zA-Z_0-9]</code>	<code>\w</code>	<code>\W</code>
digit	<code>[0-9]</code>	<code>\d</code>	<code>\D</code>

```
>>> re.findall(r"\s\d\s", "1 22 4 22 1 a b c")
[' 4 ', ' 1 ']
>>> re.findall(r"[^aeiou]", "Python Kurs")
['P', 'y', 't', 'h', 'n', ' ', 'K', 'r', 's']
```

Regular Expressions – Quantifiers

- Quantifier can be defined in ranges (min, max):

`\d{5,7}` matches sequences of 5-7 digits

- Acronym:

		Default
<code>{1}</code>	one-time occurrence	
<code>{0,}</code>	none to multiple occurrences	*
<code>{0,1}</code>	none or one-time occurrence	?
<code>{1,}</code>	at least one-time occurrence	+

```
>>> re.findall(r"[ab]{1,2}", "a aa ab ba bb b")
['a', 'aa', 'ab', 'ba', 'bb', 'b']
>>> re.findall(r"\d+", "1. Python Kurs 2012")
['1', '2012']
```

Regular Expressions – Anchors

- Anchors define special restrictions to the pattern matching:

\b word boundary, switch between \w and \W
\B negate \b
^ start of the string
\$ end of the string

```
>>> re.findall(r"^\d+", "1. Python Course 2015")  
['1']
```

- Look-around anchors (context):

- Lookahead

ab(?=c) matches "ab" if it's part of "abc"

ab(?!c) matches "ab" if not followed by a "c"

- Lookbehind

(?<=c)ab matches "ab" if it's part of "cab"

(?<!c)ab matches "ab" if not behind a "c"

Regular Expression – Rules for Pattern Matching

- Pattern analysis will start at the beginning of the string.
- If pattern matches, analysis will continue as long as the pattern is still matching (**greedy**).
- Pattern matching behavior can be changed to **non-greedy** by using the "?" behind the quantifier.
⇒ the pattern analysis stops at the first (minimal) matching

```
>>> re.findall(r"Py.*on", "Python ... Python")
['Python ... Python']
>>> re.findall(r"Py.*?on", "Python ... Python")
['Python', 'Python']
```

Regular Expressions – Groups

- `()` brackets in a pattern create a group
- Group name is numbered serially (starting with 1)
- The first 99 groups (`\1` - `\99`) can be referenced in the same pattern
- Patterns can be combined with logical **or** (`|`) inside a group

```
>>> re.findall(r"(\w+) \1", "Py Py abc Test Test")
['Py', 'Test']
>>>
>>> re.findall(r"([A-Za-z]+\d+)", "uid=2765(zdv124)")
['uid', '2765', 'zdv', '124']
>>>
>>> re.findall(r"(\.[*?\\]|<.*?>)", "[hi]s<b>sd<hal>")
['[hi]', '<b>', '<hal>']
```


Regular Expressions – Group Usage

- Some `re.*` methods return a `re.MatchObject`
⇒ contain captured groups

re_groups.py

```
text="adm06:x:706:1000:St.Graf:/home/adm06:/bin/bash"
grp=re.match(
    r"^( [a-z0-9]+ ):x:[0-9]+:[0-9]+:(.+):.+:+$",text)
if (grp):
    print("found:", grp.groups())
    print(" user ID=",grp.group(1))
    print(" name=",grp.group(2))
```

```
$ python3 re_groups.py
found: ('adm06 ', 'St.Graf')
user ID= adm06
name= St.Graf
```

Regular Expressions – Matching Flags

- Special flags can change behavior of the pattern matching
 - `re.I`: Case insensitive pattern matching
 - `re.M`: `^` or `$` will match at beginning/end of each line (not only at the beginning/end of string)
 - `re.S`: `.` also matches newline (`\n`)

```
>>> re.findall("^abc", "Abc\nabc")
[]
>>> re.findall("^abc", "Abc\nabc",re.I)
['Abc']
>>> re.findall("^abc", "Abc\nabc",re.I|re.M)
['Abc', 'abc']
>>> re.findall("^Abc.", "Abc\nabc")
[]
>>> re.findall("^Abc.", "Abc\nabc",re.S)
['Abc\n']
```

Regular Expressions – Methods (I)

findall: Simple pattern matching

⇒ list of strings (hits)

```
>>> re.findall(r"\[.*?\]", "a[bc]g[hal]def")
['[bc]', '[hal]']
```

sub: Query replace ⇒ new (replaced) string

```
>>> re.sub(r"\[.*?\]", "!", "a[bc]g[hal]def")
'a!g!def'
```

search: Find first match of the pattern

⇒ returns `re.MatchObject` or `None`

```
if re.search(r"\[.*?\]", "a[bc]g[hal]def"):
    print("pattern matched!")
```

Regular Expressions – Methods (II)

match: Starts pattern matching at beginning of the string

⇒ returns `re.MatchObject` or `None`

```
text="adm06:x:706:1000:St.Graf:/home/adm06:/bin/bash"  
grp=re.match(  
    "[a-z0-9]+):x:[0-9]+:[0-9]+:(.+)::.+$",text)
```

compile: Regular expressions can be pre-compiled

⇒ gain performance on reusing these `RegExp` multiple times

(e.g. in loops)

```
>>> pattern = re.compile(r"\[.*?\]")  
>>> pattern.findall("a[bc]g[hal]def")  
['[bc]', '[hal]']
```

Enjoy



Table of Contents

Introduction

Data Types I

Control Statements

Functions

Input/Output

Errors and Exceptions

Data Types II

Object Oriented Programming

Modules and Packages

Advanced Techniques

Tools

Regular Expressions (optional)

Summary and Outlook

Summary

We have learned:

- Multiple **data types** (e.g. „high level“)
- Common **statements**
- Declaration and usage of **functions**
- **Modules** and packages
- Errors and **Exceptions**, exception handling
- **Object oriented programming**
- Some of the often used standard modules
- Popular tools for Python developers

Not covered yet

- Closures, decorators (function wrappers)
- Meta classes
- More standard modules: mail, WWW, XML, ...
→ <https://docs.python.org/3/library>
- Profiling, debugging, unit-testing
- Extending and embedding: Python & C/C++ → <https://docs.python.org/3/extending>
- Third Party-Modules: Graphic, web programming, data bases,
... → <http://pypi.python.org/pypi>

Web Programming

- CGI scripts: Module `cgi` (standard lib)
- Web frameworks: Django, Flask, Pylons, ...
- Template systems: Cheetah, Genshi, Jinja, ...
- Content Management Systems (CMS): Zope, Plone, Skeletonz, ...
- Wikis: MoinMoin, ...



The MoinMoin Wiki Engine

Overview

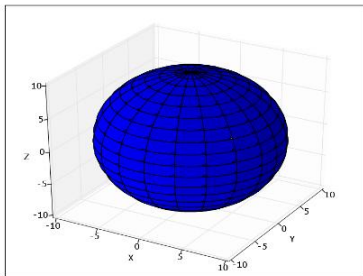
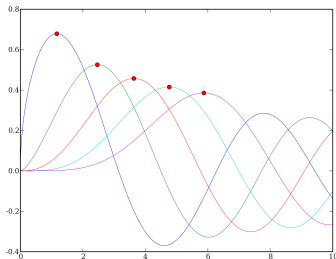
[MoinMoin](#) is an advanced, easy to use and extensible [WikiEngine](#) with a large community of users. Said in a few words, it is about collaboration on easily editable web pages. MoinMoin is Free Software licensed under the [GPL](#).

- If you want to learn more about wiki in general, first read about [WikiWikiWeb](#), then about [WhyWikiWorks](#) and the [WikiNature](#).
- If you want to play with it, please use the [WikiSandBox](#).
- [MoinMoinFeatures](#) documents why you really want to use MoinMoin rather than another wiki engine.
- [MoinMoinScreenShots](#) shows how it looks like. You can also browse *this* wiki or visit some other [MoinMoinWikis](#).

NumPy + SciPy + Matplotlib = Pylab

Alternative to MatLab:

Matrix algebra, numeric functions, plotting, ...



And more ...

- **jupyter** Notebook (interactive computational environment)
- Python IDEs
 - PyCharm
 - Eclipse (PyDev)
 - ...
- Python and other languages:
 - Jython: Python code in Java VM
 - Ctypes: Access C-libraries in Python (since 2.5 in standard lib)
 - SWIG: Access C- and C++ -libraries in Python
- **PIL**: **Python Imaging Library** for image manipulation
- **SQLAlchemy**: ORM-Framework
 - Abstraction: Object oriented access to database

Interactive High-Performance Computing with Jupyter

PRACE-Trainingskurs, online (20.04., - 22.04.202)

- Introduction to Jupyter
- Parallel computing using Jupyter
- Coupling and control of simulations
- Interactive and in-situ visualization
- Simulation dashboards
- <https://www.fz-juelich.de/SharedDocs/Termine/IAS/JSC/DE/Kurse/2021/ptc-interactive-hpc-2021.html?nn=717802>

Data Analysis and Plotting in Python with Pandas

Trainingskurs, online (27.05.2021)

- Introduction to Pandas
- Simple examples
- Hands-on exercises
- <https://www.fz-juelich.de/SharedDocs/Termine/IAS/JSC/DE/Kurse/2021/pandas-2021.html?nn=717802>

High-performance computing with Python

PRACE-Trainingskurs, online (07.06. - 11.06.2021)

- Introduces Matlab programmers to the usage of Python

- 1 Interactive parallel programming with IPython
- 2 Profiling and optimization
- 3 High-performance NumPy
- 4 Just-in-time compilation with numba
- 5 Distributed-memory parallel programming with Python and MPI
- 6 Bindings to other programming languages and HPC libraries
- 7 Interfaces to GPUs
- 8 <https://www.fz-juelich.de/SharedDocs/Termine/IAS/JSC/DE/Kurse/2021/ptc-hpc-python-2021.html?nn=717802>



PyCologne: Python User Group Köln

- Meets on the 2nd Wednesday each month at Chaos-Computer-Club Cologne
- URL: <http://pycologne.de>

Enjoy

