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 programing 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?

- Extremly 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

```python
#!/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

```python
#!/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]

[2] James Whitcomb Riley

Example: Strong and Dynamic Typing

types.py

```python
#!/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
```

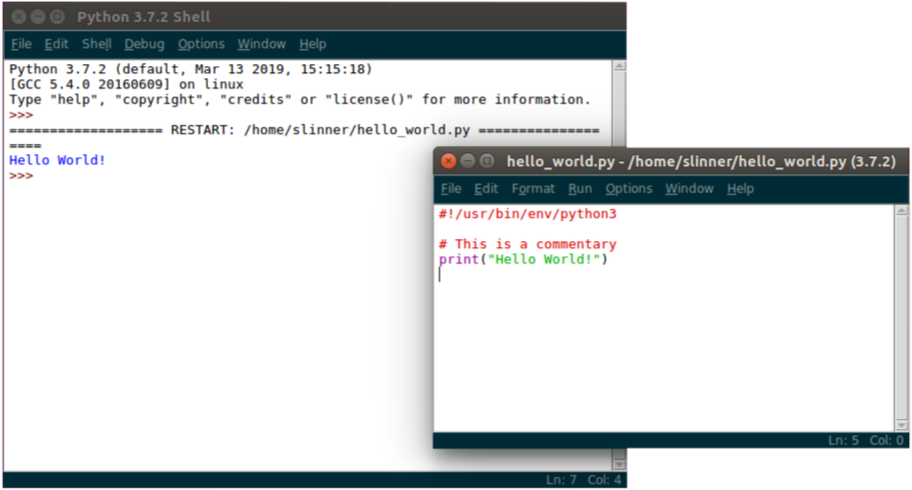
Interactive Mode

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

- **I**ntegrated **D**eve**L**opment **E**nvironment
- Part of the Python installation


Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Ln: 7 Col: 4
hello_world.py - /home/slinner/hello_world.py (3.7.2)
File Edit Format Run Options Window Help
Ln: 5 Col: 0

```
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!
>>> 
```

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

# This is a commentary
print("Hello World!")
```

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]	`#!/usr/bin/python`	`#!/usr/bin/python3`
IDLE cmd[1]	idle	idle3
print cmd (syntax)	`print`	`print()`
input cmd (syntax)	`raw_input()`	`input()`
unicode	`u"..."`	all strings
integer type	`int/long`	`int` (infinite)
...	hints in each chapter	

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

[1] linux 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**: `+`, `-`, `*`, `/`
  <u>hint:</u> *Python 2* ⇒ `1/2` = 0
  Python 3 ⇒ `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`!

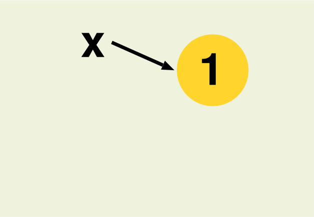
```
>>> bool([1, 2, 3])
True
>>> bool("")
False
```

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

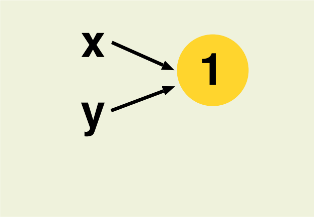
Reference - Example

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



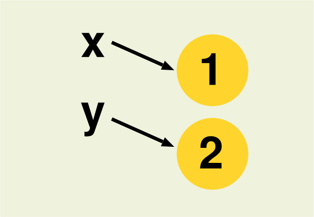
Reference - Example

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



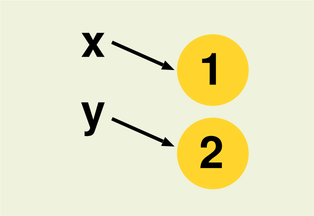
Reference - Example

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

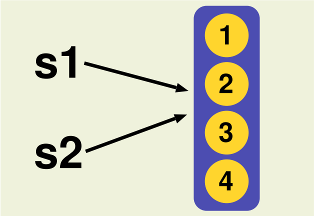


Reference - Example

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

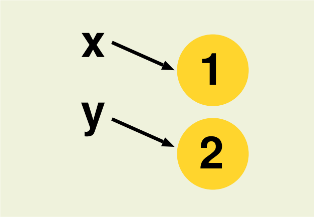


```
>>> s1 = [1, 2, 3, 4]
>>> s2 = s1
>>> s2[1] = 17
>>> s1
[1, 17, 3, 4]
>>> s2
[1, 17, 3, 4]
```

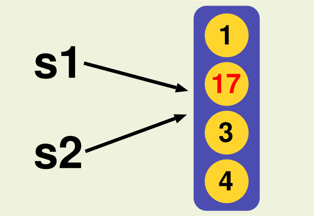


Reference - Example

```
>>> 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

python™

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

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

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

```python
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

python™

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

```python
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

```python
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:

```python
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

```python
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
m.n**f**	floating point: **m** filed size, **n** digits after the decimal point (6)
m.n**e**	floating point (exponential): **m** filed size, 1 digit before and **n** digits behind the decimal point (default: 6)
m.n**%**	percentage: similar to format **f**, *value* $*$ 100 with finalizing ’%’
m**d**	Integer number: **m** field size (**0m** ⇒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("...")` ⟺ `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

python™

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

```python
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

```python
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)

```python
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:

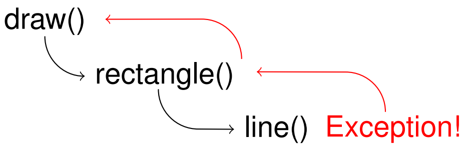
- `EnvironmentError` (`IOError` , `OSError`):
  Exception object has 3 attributes (`int` , `str` , `str`)
- Otherwise: Exception object is a string

spam_open.py

```python
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("'{0}': Invalid type".format(value))
```

The `with` Statement

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

```python
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.

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

Enjoy

python™

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