

Faculty of Computer Science, Institute of Systems Architecture, Chair of Systems Engineering

# RoboLab Autumn Course – Python #2

by M.Sc. Samuel Knobloch

# Overview

- Operators
- Tuple, List – Brief overview and differences
- Special data types: ctypes, Enum
- Testing Approaches and Unit-tests

# Operators

# Operations - Arithmetical

- Python can handle arithmetical operations:

<i>Operation</i>	<i>Example</i>	<i>Description</i>
+	res = 15 + 10	Simple addition: a + b
-	res = 15 - 10	Simple subtraction: a - b
*	res = 15 * 10	Simple multiplication: a * b
/	res = 15 / 10	Simple division: a / b
%	res = 24 % 5	Modulo operation: a % b
**	res = 4 ** 3	Exponential operation: a <sup>b</sup>
//	res = 15 // 10	Floor division: a // b

- For each operations exists a shortcut  
→ E.g. Addition: `c = a + b` → `a += b`

# Operations - Comparison

- Python can handle comparison of two variables or objects:

<i>Operation</i>	<i>Example</i>	<i>Description</i>
==	a == b	True if a equals b
!=	a != b	True if a not equals b
>	a > b	True if a greater than b
<	a < b	True if a less than b
>=	a >= b	True if a greater than or equal b
<=	a <= b	True if a is less than or equal b

# Operators – Bitwise

- Moving from objects or variables into the bit representation, we have:

<i>Operation</i>	<i>Example</i>	<i>Description</i>
&	res = a & b	Binary AND
	res = a   b	Binary OR
^	res = a ^ b	Binary XOR
~	res = ~a	Binary Complement
<<	res = a << b	Binary Shift to the left
>>	res = a >> b	Binary Shift to the right

- Example with (a = 5, b = 2):

→ Shift left:    `c = a << b`            →    `c = 0101 << 2`  
                  `c = 10100`

# Operators – Logical

- For boolean or truth values we have some options:

<i>Operation</i>	<i>Example</i>	<i>Description</i>
and	a and b	True if a and b are not zero/empty/False
or	a or b	True if a or b are not zero/empty/False
not	not a	True if a is zero/empty/False

# Operators – Further

- There are some more operations:

<b><i>Operation</i></b>	<b><i>Example</i></b>	<b><i>Description</i></b>
in	a in b	True if a is in b (e.g. b is a List)
not in	a or b	True if a is not in b (e.g. b is a List)
is	a is b	True if a and b are identical (same objects, same reference)
is not	a is not b	True if a and b are not identical (different objects, different references)



# Tuple, List – Brief overview and differences

# List

- A List is just a sequence of elements, each assigned a number (the index)
- Can contain any data type, also be mixed (see slides for Python #1)
- A List is editable and not restricted or limited to the elements on creation

- Examples:

```
→ my_list = [1, 2, 3, 4, 'five']
   cut = my_list[2:4]

print(cut) # output: [3, 4]
print(my_list[5]) # output: IndexError: list index out of range
print(my_list[4]) # output: 'five'

print(len(my_list)) # output: 5
```

# Tuple

- A Tuple is also a sequence of elements, each assigned a number (the index)
- Can contain any data type, also be mixed (see slides for Python #1)
- Compared to a List a Tuple is **immutable** and therefore limited to the elements on creation
  - We don't want the object to be changed after creation (e.g. data security reasons)
  - To extend a Tuple we have to combine it with another tuple using + operator
- Examples:
  - ```
my_tuple = (1, 2, 3, 4, 'five')  
cut = my_tuple[3:5]  
  
print(cut) # output: [4, 'five']  
print(my_tuple[5]) # output: IndexError: list index out of range  
print(my_tuple[3]) # output: 4
```

# Tuple - Good to know

- Converting a list into a tuple

```
→ my_list = [1, 2, 3, 4]
   my_tuple = tuple(my_list)
```

- Python magic for multiple return values

```
→ a = 12
   b = "Items"
```

```
return a, b # Returns a 2-tuple
```

```
return a, b, "my extra variable" # Returns a 3-tuple
```

- Combining two tuples

```
→ a = (1, 2, 3)
   b = (4, 5, 6)
   c = a + b # Result: (1, 2, 3, 4, 5, 6)
```

# Special data types: ctypes, Enum

# Special data types: ctypes

- Foreign Python library, provides C compatible data types
- Usage is really easy: Just import the data type(s) needed into your class
  - `from ctypes import * # imports everything`
- For our stack machine, we need a data type for storing a 8-bit integer (unsigned)
  - `from ctypes import c_ubyte # imports data type for uint8`
  - Usage:

```
var1 = c_ubyte(5)
var2 = c_ubyte(15)
res = c_ubyte(var1.value + var2.value)
```
- For more details see Python Documentation about this library

# Special data types: Enum

- Python's way of supporting enumerations
  - Symbolic names are bound to unique, constant values in general
- Provides some sub-classes, e.g. Enum, IntEnum, Flag, IntFlag
  - IntEnum is comparable with other integers
  - NoValue can be used from string representations
- Example:
  - ```
class MyEnum(Enum):  
    CASE1 = "something"  
    CASE2 = "another"
```
- For more details see Python Documentation about this library

# Testing Approaches and Unit-tests



# Testing Approaches

- In software development, we know different approaches on how to develop and implement
  - Test-driven development (TDD)
    - Write your tests before actually implementation the logic
    - Tests will fail initially
  - Acceptance test-driven development (ATDD)
    - Same as TDD, but more for communication between developers and customer
  - Behaviour-driven development (BDD)
    - Combination of TDD and ATDD
    - Used to describe a specific workflow or behaviour instead of testing a single unit
- In this course we use the TDD approach

# Unit-tests - Basics

- Used to test a specific functionality or single method
  - Specific input and expected output parameters are well know
- In Python, we use the standard library „unittest“
  - `import unittest`
- To execute test on the console we have some options:
  - `python3 -m unittest test.py # Execute every test in file`  
`python3 -m unittest test.MyTestClass # Execute only tests in MyTestClass`
- Besides unit-tests, there are more complex types for testing including:
  - Integration-tests, Validation-tests, Component-tests / Module-tests, Stress-tests, Performance-tests

# Unit-tests – Classes and Definitions

- The most commonly used class is “**TestCase**”
  - We inherit from this class while defining our own tests:

```
class MyTestClass(unittest.TestCase):
```
  - Beside `TestCase`, there are some more classes, e.g. ***FunctionTestCase***, ***TestSuite***
- For writing a valid test case we have to prefix our function with “**test\_**” to be recognized correctly
  - Functions without this prefix will be ignored and not considered as a test
- The library `unittest` comes with a lot of assert-routines for checking the results and return values
  - `assertEqual()`, `assertTrue()`, `assertFalse()`, `assertIsInstance()`, ...
  - For more and additional information about these functions see [Python Documentation](#)

# Unit-tests – Example

- Given our well-known class *HammingCode*, an example for an instance test would look like this

→

```
import unittest
from hamming_code import HammingCode

class TestHammingCode(unittest.TestCase):

    def test_instance(self):
        hc = HammingCode()

        self.assertIsInstance(hc, HammingCode)
```

# Sources

- [https://www.tutorialspoint.com/python/python\\_basic\\_operators.htm](https://www.tutorialspoint.com/python/python_basic_operators.htm)
- [https://www.tutorialspoint.com/python/python\\_lists.htm](https://www.tutorialspoint.com/python/python_lists.htm)
- [https://www.tutorialspoint.com/python/python\\_tuples.htm](https://www.tutorialspoint.com/python/python_tuples.htm)
- <https://docs.python.org/3.8/library/ctypes.html>
- <https://docs.python.org/3.8/library/enum.html>
- <https://docs.python.org/3.8/library/unittest.html>