1 – Python basics

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

Bremen Center for Computational Materials Science

Universität Bremen

https://www.bccms.uni-bremen.de/people/b-aradi/wissen-progr/python/2023

Outline

- About Python
- Basic (scalar) data types
- Control structures

Python

- Invented/Created by Guido von Rossum 1989
- Has a huge community
- De facto standard script language for scientific applications (though Julia is becoming a possible alternative)
- Python is an **interpreted** language
  - Fast development (less code, no compilation necessary)
  - Often much slower than compiled languages (though, speed critical parts can be written in C/C++/Fortran)

Python 3

- **actively developed**
- “cleaned up” version of Python 2
- Introduced backwards incompatible changes

Python 2

- **Deprecated (**support ended in 2020)
- **don’t use it for new projects**

Learning Python

Internet

- Official Python documentation, especially Tutorial and Library Reference: https://docs.python.org/3/
- Real Python
- Dive into Python (for advance learner, very good for OO-concepts)
- Newsgroups, mailing lists, stackoverflow, etc.
- :

Books

- M. Lutz: Learning Python (very-very detailed)
- M. Lutz: Programming Python (programming techniques)
- L. Ramalho: Fluent Python (advanced level)
- :

Data types

Immutable data types

- Can not be changed once they have been created
- You must create a new (changed) instance if you want to change them
- Examples: bool (True, False), integer, float, string, tuple, frozen set, etc.

Mutable data types

- Their content can be changed after their creation
- Examples: list, set, dictionary, file, etc.
- Handling of mutable data types can have certain “**side-effects**”

Integers (int)

- Range: arbitrary
- If value is beyond the long int data type in C (2**63 on 64 bit machines), operations become rather slow (runs emulated, not natively)

Runs the cell a given amount of time and measures execution time

% Jupyter kernel “magic” commands

```
%%timeit -r 10
num = 2**3625
for ii in range(63):
    num *= 2
```

Compare

```
%%timeit -r 10
num = 2**0
for ii in range(63):
    num *= 2
```

Floating point numbers (float, complex)

Real numbers

- The same as double type in C
  - Range: +/-1E-323 – +/-1E+308
  - Precision: 16 digits
- Can be entered either in fixed or in expontential notation

```
>>> 0.123
0.123
>>> 1.23E-1
0.123
>>> 9e-1300
0
>>> 9e1000
inf
```

Complex numbers

- Represented by a pair of real numbers
- Real and imaginary part have the same range then usual real numbers
- Input as ***RealPart + ImaginaryPartJ***

```
>>> 2.0 + 3.3j
(2+3.3j)
```

Arithmetic operators

+	Addition
-	Substraction
*	Multiplication
/	Division
//	Integer division
%	Division remainder
-	Negation
**	Power

```
>>> 1 + 2
3
>>> 3 - 4
-1
>>> 5 * 6
30
>>> 5 / 2
2.5
```

```
>>> 5 // 2
2
>>> 5 % 2
1
>>> -8
-8
>>> 2**0.5
1.4142135623730951
```

Relation operators

```
==    equal
!=    unequal
<     less
<=    less equal
>     greater
>=    greater equal
```

```
>>> 3 == 2
False
>>> 3 != 2
True
>>> 3 < 2
False
>>> 3 > 2
True
>>> 3 >= 2
True
>>> 3 <= 2
False
```

Comparison gives bool type as result (True/False)

```
>>> 3.0+2j == 3.0+3j
False
>>> 3.0+2j != 3.0+3j
True
>>> 3.0+2j < 2.0-1.2j
Traceback (most recent call...
```

Error: Complex numbers can not be ordered

Comparing with == or != is OK

Booleans (bool) & logical operators

- They are actually numbers, only shown differently
  - **False**: 0, **True**: 1

```
>>> True
True
>>> False
False
>>> 2 * True
2
```

Logical operators

- Logical **AND** (True if both operands True)
- Logical **OR** (True if any of the operands True)
- Logical **NOT** (Negates operand)

```
>>> True and False
False
>>> False or True
True
>>> not True
False
```

- In Python each object can serve as a logical value (details later)

Assignment

- An object (e.g. result of an operation) gets a name assigned (variable name)

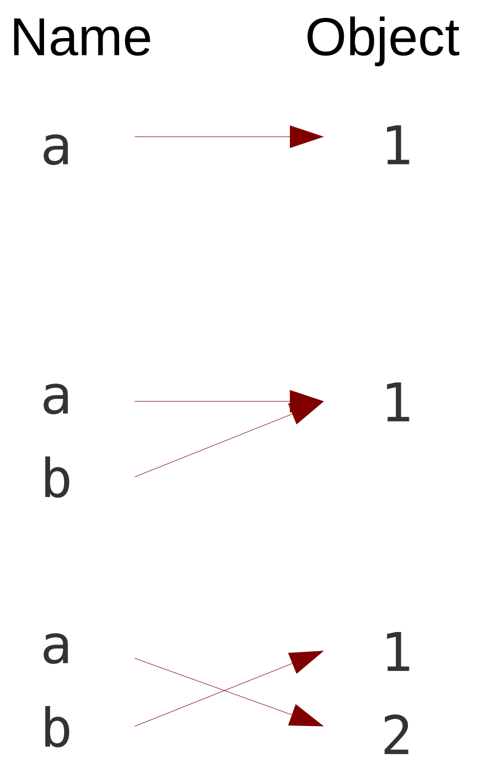
- **Name = Object**
  Name points to / aliases Object

- **Name1 = Name2**
  Name1 points to the same object which Name2 points to

- When using a variable name in an expresssion, it will be substituted with the object it points to.

- There are **no “classic” variables** in Python, just **pointers / aliases**!

```
>>> a = 1
>>> a
1
>>> b = a
>>> b
1
>>> a = 2
>>> a
2
>>> b
1
>>> a + b
3
```



Strings

- Strings are specified between apostrophes or quotes:

```
>>> name1 = 'john'
>>> name2 = "tom"
>>> name1
'john'
>>> name2
'tom'
```

- Length of a string can be queried by the **len()** function:

```
>>> len(name1)
4
```

- Multilne strings can be specified between triple apostrophes or quotes:

```
>>> longstr = """First line
... followed by the second"""
>>> longstr
'First line\nfollowed by the second'
```

newline character

Strings

- Parts of a string can be accessed by the **[]** operator:

Elements are enumerated **starting with zero**

When selecting ranges as **[lower:upper]**, the **lower bound** is **inclusive** the **upper bound** is **exclusive**

Range increment can be also specified with **[lower:upper:increment]**

When lower bound is omitted, range starts from the very first element (0 – range increment pos., last – range increment neg.)

When upper bound is omitted, range ends beyond last element (last element is included)

Negative range increment: iterating backwards

Empty range returns empty string

```
>>> txt = "some text"
>>> txt[0]
's'
>>> txt[0:4]
'some'
>>> txt[0:9:2]
'sm et'
>>> txt[:4]
'some'
>>> txt[4:]
' text'
>>> txt[8:4:-1]
'txet'
>>> txt[3:3]
''
```

Strings

- Strings are immutable, they can not be changed once created:

```
>>> txt[0] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str'...does not support item assignment
```

- Strings can be **concatenated** by the **+ operator** or by whitespace for string literals:

```
>>> name1 + " " + name2
'john tom'
>>> "str1" "str2"
'str1str2'
```

- Strings can be **repeated** by the *** operator**:

```
>>> "ab" * 3
'ababab'
```

String formatting, f-strings

- Formatted strings (**f-strings**): String scontaining expressions with optional formatting options
- Expressions are enclosed in {}

```
aa = 12
bb = 135
print(f"a = {aa}, b = {bb}")
```

```
a = 12, b = 135
```

- Optional formatting options can be specified after the expression, separated by a colon (**:**)

```
print(f"a = {aa:3d}\nb = {bb:3d}")
```

Newline character — Field width — Data type

```
a =  12
b = 135
```

- Data type must match expression type:

```
cc = 12.35
print(f"c = {cc:4d}")
```

→

```
ValueError: Unknown format code 'd'
for object of type 'float'
```

Few formatting options

:*W*d	integer number
:*W*.*P*f	floating point number in fixed notation
:*W*.*P*e	floating point number in exponential notation (with small e)
:*W*.*P*E	floating point number in exponential notation (with capital E)
:*W*.*P*g	:f or :e depending on the value of the floating point
:*W*.*P*G	:f or :e depending on the value of the floating point
:Ws	string (converts given object to a string)

W (width) minimal field width (optional)

P (precision) number of decimal places (optional)

```
ff = 1.2
f"{ff:12.4E}"
f"{ff:12E}"
f"{ff:.4E}"
ss = "ab"
f"{ss:5s}"
```

→

```
'  1.2000E+00'
'1.200000E+00'
'1.2000E+00'
```

Numbers aligned right

```
'ab   '
```

String aligned left

For further formatting options see the format specification mini-language

Few remarks on string formatting

- If the field with is too small for the given represenation, it will be automatically expanded

```
num = 123
f"|{num:1d}|"
```

→ `'|123|'`

- If you need literal curly braces in the formatted string, they must be doubled:

```
num = 122
f"{{{0:d}}}"
```

→ `'{123}'`

- Formatted strings can be created with the **.format()** method as well
- Expressions are given as parameters of the **.format()** method

```
num = 123
"|{:4d}|".format(num)
```

→ `'|  123|'`

- Parameters can be refered by their position

```
num1 = 12
num2 = 34
"|{0:d}, {1:d}, {0:d}|".format(num1, num2)
```

(0 → num1, 1 → num2)

`'|12, 34, 12|'`

Converting data types into each other

- Each data type has a special function, which tries to convert its argument into an object with the given data type:

```
int(), float(), complex(), str()
```

- Argument can have arbitrary data type
- If the conversion fails, an exception is raised (error)

```
>>> int(3.2)
3
>>> float("12.1")
12.1
>>> complex("3+2j")
(3+2j)
>>> complex("3.0+2.0j")
(3+2j)
```

```
>>> valstr = "3"
>>> int(valstr)
3
>>> int("hello")
Traceback ...ValueError: ...
```

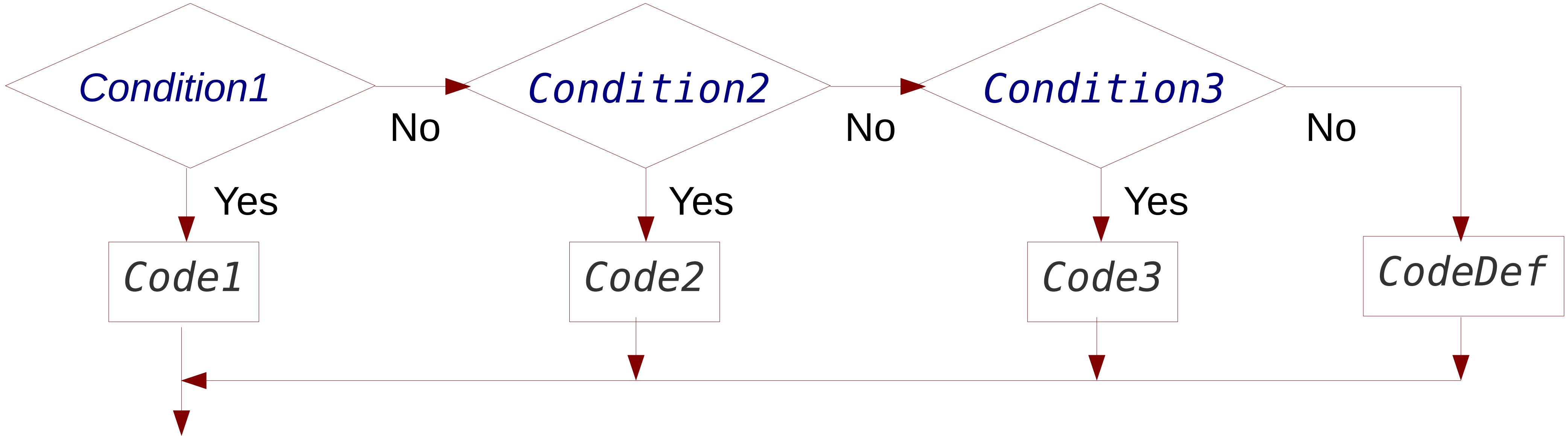
Branching

- Optional code execution based on condition evaluation
- Nested blocks in Python **start with colon (:)**
- One should always use **4 spaces as indentation**
- End of nested block is signalised by an **unindented statement**

Indentation signalises nesting

```
if Condition1:
    Code1
elif Condition2:
    Code2
elif Condition3:
    Code3
else:
    CodeDef
```

Start of a nested block



Indentation in Python

- Indentation is not optional, but part of the language semantics
- Indentation signalises nesting
- Amount of indentation signalises nesting depth
- Each nested block should be indented by exactly 4 space characters
- Inconsistent indentation leads either to syntax error or to wrong code logics

```
if answer[0] == "y":
    print("OK, you agree")
else:
    print("I see")
    print("You don’t agree")
print("Let’s continue")
```

Indented, belongs to if-block
(Only executed if answer[0] == “y”)

Indented, belongs to else-block
(Only executed if answer[0] != “y”)

Unindented, outside of if/else block
(Always executed)

- Use an editor which supports Python to ensure proper indentation!

If-else expression

- One can choose between two expressions with an if/else construct within an expression
- Use it only for trivial (short) cases

Syntax:

```
true_expression if condition else false_expression
```

```
mytype = "pos. semidef" if b >= 0 else "negative"
print("b is of type:", mytype)
```

Evaluation as bool expression

- Each object can be evaluated as a bool expression
- Evaluation is type dependent: Numerical types are usually False, if their value is zero. Container types are usually False, if they are empty

Object type	Evaluated to False	Evaluated to True
bool	False	True
int	0	any other value
float	0.0	any other value
complex	0.0+0.0j	any other value
string	"" (empty string)	contains at least one char.
list	[] (empty list)	contains at least one element
dict	{} (empty dict)	contains at least one element

```
if num % 2:
    print("odd")
```

if num % 2 != 0

```
if not num % 2:
    print("even")
```

if num % 2 == 0

for loop

- Iteration over given values can be realised with a **for-loop**

```
for loop_variable in iterable_object:
    loop code
```

Use the **for** loop, if the nr. of iterations is **known** in advance

- The iterable object can be anything, which is able to return values one-by-one (implements the iterator-interface)
- Example: string is iterable, it returns its characters one by one:

```
name1 = 'john'
for char in name1:
    print("Char: ", char)
```

```
Char:  j
Char:  o
Char:  h
Char:  n
```

- If loop variable is not needed, use _ as a placeholder:

```
for _ in range(4):
   tt.left(90)
   tt.forward(10)
```

Loop variable not needed within the loop

Range iterator

- The **range()** function returns an iterator over integers

```
range(from, to, step)
```

- Lower bound is included, upper bound is excluded (as for substring ranges)

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

- If step size is omitted, step is is assumed to be 1

```
range(0, 4)          →   [0, 1, 2, 3]
```

- If **range()** is called with one argument, it is interpreted as upper bound

```
range(4)             →   [0, 1, 2, 3]
```

- If selected range is empty, iterator does not return any values

```
range(4, 4)          →   []
```

Note: You can use the list constructor to explicitly show the values yielded by an iterator:

```
list(range(4))
```

for loop: break, continue

- The break and continue statements can be also used within a for-loop
  - **break**: Terminates loop execution a continues after loop-block
  - **continue**: Jumps to loop header and iterates over next item

```
for num in range(4, 8):
    if not num % 5:
        break
print("First number divisible by 5:", num)
```

```
Num:  5
```

```
print("All numbers not divisable by 5:")
for num in range(4, 8):
    if not num % 5:
        continue
    print(num)
```

```
4
6
7
```

for loop: else

- The **else** branch of a for-loop is executed, if the loop terminated after having iterated over all elements (and not due to a break statement)

```
for num in range(6, 10):
    if not num % 5:
        break
else:
    print("No multiple of 5")
```

↔ Equivalent code

```
found = False
for num in range(6, 11):
    if not num % 5:
        found = True
        break
if not found:
    print("No multiple of 5")
```

while loop

Use the **while** loop, if the nr. of iterations is **not known** (or is difficult to determine) in advance

- Repeats a program block as long a condition is fulfilled

```
while Condition:
    Loop code
```

- If the condition is not fullfilled (any more), code execution continues after the while-block

```
num = 1
while num <= 20:
    print(num)
    num = num * 2
print("First above 20: ", num)
```

→

```
1
2
4
8
16
First above 20: 32
```

while loop: break, continue

- Execution order in loops can be modified:
  - **break**: terminates loop and continues execution after loop block
  - **continue**: jumps back to loop header and evaluates loop condition again

```
while True:
    answer = input("Do you agree (y/n)? ")
    if answer != "y" and answer != "n":
        print("Invalid answer! Try it again!")
        continue
    if answer == "y":
        print("Good answer, thanks!")
        break
    print("Valid answer, but I don't like it!")
print("Nice that we agree!")
```

Reads console input as string

Endless loop, should be exited via break at some point

while loop: else

- Optional else-branch of a while loop is executed, if the loop execution was aborted due to loop condition becoming False (and not due to a break statement)

```
ii = 0
while ii < 5:
    ii += 1  # ii = ii + 1
    answer = input("Do you agree? (y/n) ")
    if answer == "y" or answer == "n":
        break
else:
    print("Too many invalid answers, I'll assume yes.")
    answer = "y"
print("Your answer was: ", answer)
```

Have fun!