CFDS® – Chartered Financial Data Scientist
Introduction to Python

Prof. Dr. Natalie Packham

27 November 2024

Table of Contents

1 Data types and structures

1.1 First examples

First examples

- Printing to the standard output:

In [ ]:

```python
print("Hello World")
```

First examples

- A calculator:

In [ ]:

```python
2+4
```

In [ ]:

```python
2-4
```

In [ ]:

```python
2**12
```

- A double asterisk (`**`) is the operator for taking powers.
- The code in a cell is executed by clicking on "Run" or via the shortcut `Shift + Enter`.

1.2 Variables

Variables

- The kernel stores data in variables. The assignment operator `=` assigns a variable name (left-hand side) a value (right-hand side).

In [ ]:

```python
a = 5
b = 2
a*b
```

In [ ]:

```python
savings = 50
```

In [ ]:

```python
savings
```

Variables

- Calculate the savings value in seven years, given an interest rate of 7%.

In [ ]:

```python
r = 0.07
future_value = savings*(1+r)**7
future_value
```

Built-in functions

- A number of built-in functions are available ([link (https://docs.python.org/3/library/functions.html)](https://docs.python.org/3/library/functions.html)):

		Built-in Functions		
`abs()`	`delattr()`	`hash()`	`memoryview()`	`set()`
`all()`	`dict()`	`help()`	`min()`	`setattr()`
`any()`	`dir()`	`hex()`	`next()`	`slice()`
`ascii()`	`divmod()`	`id()`	`object()`	`sorted()`
`bin()`	`enumerate()`	`input()`	`oct()`	`staticmethod()`
`bool()`	`eval()`	`int()`	`open()`	`str()`
`breakpoint()`	`exec()`	`isinstance()`	`ord()`	`sum()`
`bytearray()`	`filter()`	`issubclass()`	`pow()`	`super()`
`bytes()`	`float()`	`iter()`	`print()`	`tuple()`
`callable()`	`format()`	`len()`	`property()`	`type()`
`chr()`	`frozenset()`	`list()`	`range()`	`vars()`
`classmethod()`	`getattr()`	`locals()`	`repr()`	`zip()`
`compile()`	`globals()`	`map()`	`reversed()`	`__import__()`
`complex()`	`hasattr()`	`max()`	`round()`	

Built-in functions

In [ ]:

```
round(future_value, 4)
```

Use `help()` to learn more about functions:

In [ ]:

```
help(round)
```

1.3 Basic Data Types

Basic Data Types

Object type	Meaning	Used for
int	Integer value	Natural numbers
float	Floating-point number	Real numbers
bool	Boolean value	Something true or false
str	String object	Character, word, text

Integers

- Zero, natural numbers and their negative counterparts, i.e., $\mathbb{Z} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$

In [ ]:

```python
a = 4
type(a)
```

Floating point numbers ("floats")

- Computer representation of real numbers, i.e., $\mathbb{R}$

In [ ]:

```python
1.6 / 4
```

In [ ]:

```python
type(1.6/4)
```

Integers and floats

In [ ]:

```python
int(5)
```

In [ ]:

```python
float(5)
```

In [ ]:

```python
type(5)
```

In [ ]:

```python
type(5.)
```

Boolean: True or False

- Data type associated with logical expressions
- A Boolean is either `True` or `False`

In [ ]:

```python
type(5>4)
```

In [ ]:

```python
5 == 3
```

In [ ]:

```python
5 != 3
```

Booleans

In [ ]:

```python
5 < 3
```

In [ ]:

```python
5 > 3
```

In [ ]:

```python
5 >= 3
```

In [ ]:

```python
5 <= 3
```

Integer representation of Booleans

In [ ]:

```python
int(True)
```

In [ ]:

```python
int(False)
```

Logical operators: `and, or`

In [ ]:

```python
True and True
```

In [ ]:

```python
True and False
```

In [ ]:

```python
False and False
```

In [ ]:

```python
True or False
```

Strings

- Strings represent text, i.e., a string is a sequence of characters.
- The basic object to represent strings is `str`.
- A string object is defined by wrapping its contents in in single or double quotation marks.

In [ ]:

```python
t = 'this is a string object'
```

In [ ]:

```python
t
```

In [ ]:

```python
t = "this is a string object"
t
```

- Simple operations can be achieved with built-in string functions:

In [ ]:

```python
t.capitalize()
```

In [ ]:

```python
t.split()
```

- `\n` is used to force a new line.
- Note: Writing `t.` and hitting the `TAB` key shows the available built-in functions.

Useful string methods

Method	Arguments	Returns/result
capitalize	()	Copy of the string with first letter capitalized
count	(*sub*[, *start*[, *end*]])	Count of the number of occurrences of substring
decode	([*encoding*[, *errors*]])	Decoded version of the string, using *encoding* (e.g., UTF-8)
encode	([*encoding*+[, *errors*]])	Encoded version of the string
find	(*sub*[, *start*[, *end*]])	(Lowest) index where substring is found
join	(*seq*)	Concatenation of strings in sequence *seq*
replace	(*old, new*[, *count*])	Replaces *old* by *new* the first *count* times
split	([*sep*[, *maxsplit*]])	List of words in string with *sep* as separator
splitlines	([*keepends*])	Separated lines with line ends/breaks if *keepends* is True
strip	(*chars*)	Copy of string with leading/lagging characters in *chars* removed
upper	()	Copy with all letters capitalized

Source: Python for Finance, 2nd ed.

Printing

- Use `print()` to print strings and other objects.

In [ ]:

```python
print('Python for Finance')
```

In [ ]:

```python
print(t)
```

Printing

In [ ]:

```python
i = 0
while i < 4:
    print(i)
    i += 1
```

In [ ]:

```python
i = 0
while i < 4:
    print(i, end='|')
    i += 1
```

Formatted printing

- The syntax with curly braces ({}) was introduced in Python 3.7.

In [ ]:

```python
'this is an integer %d' % 15
```

In [ ]:

```python
'this is an integer {:d}'.format(15)
```

In [ ]:

```python
'this is an integer {:4d}'.format(15)
```

Formatted printing

In [ ]:

```python
'this is a float %f' % 15.3456
```

In [ ]:

```python
'this is a float %.2f' % 15.3456
```

In [ ]:

```python
'this is a float {:f}'.format(15.3456)
```

In [ ]:

```python
'this is a float {:.2f}'.format(15.3456)
```

Another useful way to print is to concatenate strings with a + sign:

In [ ]:

```python
'this is a float ' + str(15.3456)
```

1.4 Basic Data Structures

Basic Data Structures

Object type	Meaning	Used for
tuple	Immutable container	Fixed set ob objects, record
list	Mutable container	Changing set of objects
dict	Mutable container	Key-value store
set	Mutable container	Collection of unique objects

- We shall skip dictionaries and sets here as they will be rarely used (if at all) once we have introduced `pandas`.
- Similarly, we shall just skim over lists as they are often used as an auxiliary data structure, e.g. in loops.

Tuples

- The elements of a tuple are written between parentheses, or just separated by commas.
- Tuples are immutable.

In [ ]:

```python
t = 1, 2, 3, 'Tom'
t
```

In [ ]:

```python
t = (1, 2, 3, 'Tom')
t
```

In [ ]:

```python
t[3]
```

> NOTE: Indices start at 0, not at 1.

Lists

- A list is created by listing its elements within square brackets, separated by commas.

In [ ]:

```python
l = [1.2,"john", "timmy", "helena"]

l
```

In [ ]:

```python
l[2]
```

In [ ]:

```python
type(l)
```

> NOTE: Indices start at 0, not at 1.

Lists

- Negative indices: Count from the end

In [ ]:

```python
l
```

In [ ]:

```python
l[-1]
```

In [ ]:

```python
l[-2]
```

Lists

- Extract elements using `[from:to]`:

In [ ]:

```python
l
```

In [ ]:

```python
l[1:3]
```

> NOTE: names[from:to] contains the elements from "from" to ("to"-1).

Lists

- Extract sublists, e.g. from index 2 or until index 2:

In [ ]:

```python
l[2:]
```

In [ ]:

```python
l[:2]
```

Lists

- Extract sublists by using `[from:to:step]`, e.g. every second element or all elements from index 2:

In [ ]:

```python
l
```

In [ ]:

```python
l[::2]
```

In [ ]:

```python
l[2::]
```

Lists

- Use `[from:to:step]` to reverse a list:

In [ ]:

```python
l[::-1]
```

Lists

- Use `*` and `+` to repeat and join lists:

In [ ]:

```python
l*2
```

In [ ]:

```python
l + l[::-1]
```

Lists

- Lists are mutable

In [ ]:

```python
l
```

In [ ]:

```python
l[0] = "thomas"
```

In [ ]:

```python
l
```

In [ ]:

```python
l[2:4] = ["hannah", "laura"]
```

In [ ]:

```python
l.append([4,3])
```

In [ ]:

```python
l
```

List methods

- To view the methods available to a list, type the list name with a full-stop (e.g. `l.`)and press the `Tab` button.

Method	Arguments	Returns/result
`l[i] = x`	`[i]`	Replaces i-th element by x
`l[i:j:k] = s`	`[i:j:k]`	Replaces every k-th element from i to $j-1$ by s
`append`	`(x)`	Appends x to object
`count`	`(x)`	Number of occurrences of object x
`del l[i:j:k]`	`[i:j:k]`	Deletes elements with index values i to $j-1$
`extend`	`(s)`	Appends all elements of s to object
`index`	`(x[, i[, j]])`	First index of x between elements i and $j-1$
`insert`	`(i, x)`	Inserts x at/before index i
`remove`	`(i)`	Removes element with index i
`pop`	`(i)`	Removes element with index i and returns it
`reverse`	`()`	Reverses all items in place
`sort`	`([cmp[, key[, reverse]]])`	Sorts all items in place

Source: Python for Finance, 2nd ed.

1.5 Control flow

Control flow

- Control flow refers to control structures such as `if`-statements and `for`-loops that control the order in which code is executed.

Control flow: `if`-statements

- The `if/elif/else` statement executes code based on whether a condition is true or false:

In [ ]:

```python
if 2+2 == 4:
    print("I am a math genius!")
```

- The condition `2+2==4` resolves to true, so the indented code following the condition is executed.
- Code blocks are delimited by indentation.

Control flow: `if`-statements

- In the code below the conditions are checked and the first block matching a `TRUE` condition is executed:

In [ ]:

```python
x = 100
if x == 1:
    print(x)
elif x == 2:
    print(x)
elif x == 3:
    print(x)
elif x == 4:
    print(x)
else:
    print("{} > 4".format(x))
```

Control flow: `for`-loops

A `for` loop iterates over a sequence of values and executes the respective code block:

In [ ]:

```python
names = ["John", "Alina", "Timmy", "Helena"]
for participant in (names):
    print('{} is taking this lecture.'.format(participant))
```

- If the sequence is a counter, i.e., a sequence of numbers, then the function `range()` is useful:

In [ ]:

```python
for (i) in range(4):
    print(i)
```

List comprehension allows to generate lists in a one-line `for` statement:

In [ ]:

```python
l=[i for i in range(4)]
l
```

Control flow: `while`-statement

- The `while` loop repeatedly executes a block of statements provided the condition remains `TRUE`:

In [ ]:

```python
a = 4
i = 0
while i < a:
    print("{} is smaller than {}." .format(i, a))
    i = i+1
```

- The `break` keyword interrupts control flow:

In [ ]:

```python
for i in range(5):
    print("{} is smaller than {}".format(i, 3))
    if i == 2:
        break
```

1.6 Functions

Functions

- It is useful to group code that is repeatedly used in **functions**.
- A `function` -block consists of the function header and the function body.
- As with other code blocks, the function body must be indentend.

In [ ]:

```python
def circle_circumference(radius):
    return 2 * 3.14159 * radius * 2
```

In [ ]:

```python
circle_circumference(5)
```

Function syntax

- The syntax for writing a function is:
  - keyword `def` followed by
  - function's name followed by
  - the arguments of the function, separated by commas, in parentheses followed by a colon (`:`)
  - function body (indented)
  - optional: within the function body the `return` keyword can be used to return an object or value

Function syntax

- Function arguments are passed "by reference" meaning that changes to an object can be made within a function:

In [ ]:

```python
def try_to_modify(x, y, z):
    x = 23
    y.append(42)
    z = [99] # new reference
    print(x)
    print(y)
    print(z)

a = 77 # immutable variable
b = [99] # mutable variable
c = [28]

try_to_modify(a, b, c)
print(a)
print(b)
print(c)
```

Function syntax

- Function arguments can be given default values:

In [ ]:

```python
def plus10(x = 50):
    return x + 10
```

In [ ]:

```python
plus10()
```

In [ ]:

```python
plus10(20)
```