Computational Finance and FinTech
Data Types and Structures

Contents

0 Data types and structures

- Further reading: **Py4Fi, Chapter 3**

0.1 First examples

First examples

- Printing to the standard output:

```
[1]: print("Hello World")
```

```
Hello World
```

First examples

- A calculator:

```
[2]: 2+4
```

```
[2]: 6
```

```
[3]: 2-4
```

```
[3]: -2
```

```
[4]: 2**12
```

```
[4]: 4096
```

- 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`.

0.2 Variables

Variables

- Variables store data. The assignment operator = assigns a variable name (left-hand side) a value (right-hand side).

```
[5]:  a = 5
      b = 2
      a*b
```

[5]: 10

```
[6]:  savings = 50
```

```
[7]:  savings
```

[7]: 50

Variables

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

```
[8]:  r = 0.07
      future_value = savings*(1+r)**7
      future_value
```

[8]: 80.28907382392153

Built-in functions

- A number of built-in functions are available (link):

		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

Built-in functions

```
[9]:  round(future_value, 4)
```

[9]: 80.2891

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

```
[10]:  help(round)
```

```
Help on built-in function round in module builtins:

round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None.  Otherwise
    the return value has the same type as the number.  ndigits may be negative.
```

0.3 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., $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$

```
[11]: a = 4
      type(a)
```

```
[11]: int
```

Floating point numbers ("floats")

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

```
[12]: 1.6 / 4
```

```
[12]: 0.4
```

```
[13]: type(1.6/4)
```

```
[13]: float
```

Integers and floats

```
[14]: int(5)
```

```
[14]: 5
```

```
[15]: float(5)
```

```
[15]: 5.0
```

```
[16]: type(5)
```

```
[16]: int
```

```
[17]: type(5.)
```

```
[17]: float
```

Boolean: True or False

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

```
[18]: type(5>4)
```

```
[18]: bool
```

```
[19]: 5 == 3
```

```
[19]: False
```

```
[20]: 5 != 3
```

```
[20]: True
```

Booleans

```
[21]: 5 < 3
```

```
[21]: False
```

```
[22]: 5 > 3
```

```
[22]: True
```

```
[23]: 5 >= 3
```

```
[23]: True
```

```
[24]: 5 <= 3
```

```
[24]: False
```

Integer representation of Booleans

```
[25]: int(True)
```

```
[25]: 1
```

```
[26]: int(False)
```

```
[26]: 0
```

Logical operators: and, or

```
[27]: True and True
```

```
[27]: True
```

```
[28]: True and False
```

```
[28]:  False
```

```
[29]:  False and False
```

```
[29]:  False
```

```
[30]:  True or False
```

```
[30]:  True
```

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.

```
[31]:  t = 'this is a string object'
```

```
[32]:  t
```

```
[32]:  'this is a string object'
```

```
[33]:  t = "this is a string object"
       t
```

```
[33]:  'this is a string object'
```

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

```
[34]:  t.capitalize()
```

```
[34]:  'This is a string object'
```

```
[35]:  t.split()
```

```
[35]:  ['this', 'is', 'a', 'string', 'object']
```

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

Useful string methods

Source: Python for Finance, 2nd ed.

Printing

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

```
[36]:  print('Python for Finance')
```

```
       Python for Finance
```

```
[37]:  print(t)
```

```
       this is a string object
```

Printing

5

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

Useful string methods

```
[38]: i = 0
while i < 4:
    print(i)
    i += 1
```

```
0
1
2
3
```

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

```
0|1|2|3|
```

Formatted printing

- The syntax with curly braces ({}) is new in Python 3.7.

```
[40]: 'this is an integer %d' % 15
```

```
[40]: 'this is an integer 15'
```

```
[41]: 'this is an integer {:d}'.format(15)
```

```
[41]: 'this is an integer 15'
```

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

```
[42]: 'this is an integer   15'
```

Formatted printing

```
[43]: 'this is a float %f' % 15.3456
```

```
[43]:  'this is a float 15.345600'
```

```
[44]:  'this is a float %.2f' % 15.3456
```

```
[44]:  'this is a float 15.35'
```

```
[45]:  'this is a float {:f}'.format(15.3456)
```

```
[45]:  'this is a float 15.345600'
```

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

```
[46]:  'this is a float 15.35'
```

Formatted printing

- Here is an example on how to use formatted printing in a loop

```
[47]:  i = 0
       while i < 4:
           print('the number is {:d}'.format(i))
           i += 1
```

```
the number is 0
the number is 1
the number is 2
the number is 3
```

Formatted printing

- More examples:

```
[48]:  print("My savings are almost depleted, I have only {} dollars left.".
        →format(savings))
```

```
My savings are almost depleted, I have only 50 dollars left.
```

```
[49]:  print("{first} is my native language, but I also speak {second} and {third}.".
        →format(first="German", second="English", third="French"))
```

```
German is my native language, but I also speak English and French.
```

Note: To learn more about string replacements, see the Python Documentation.

0.4   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

Tuples

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

```
[50]: t = 1, 2, 3, 'Tom'
      t
```

```
[50]: (1, 2, 3, 'Tom')
```

```
[51]: t = (1, 2, 3, 'Tom')
      t
```

```
[51]: (1, 2, 3, 'Tom')
```

```
[52]: t[3]
```

```
[52]: 'Tom'
```

NOTE: Indices start at 0, not at 1.

Lists

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

```
[53]: l = [1.2,"john", "timmy", "helena"]

      l
```

```
[53]: [1.2, 'john', 'timmy', 'helena']
```

```
[54]: l[2]
```

```
[54]: 'timmy'
```

```
[55]: type(l)
```

```
[55]: list
```

NOTE: Indices start at 0, not at 1.

Lists

- Negative indices: Count from the end

```
[56]: l
```

```
[56]: [1.2, 'john', 'timmy', 'helena']
```

```
[57]: l[-1]
```

```
[57]: 'helena'
```

```
[58]: l[-2]
```

```
[58]: 'timmy'
```

Lists

- Extract elements using [from:to]:

```
[59]: l
```

```
[59]: [1.2, 'john', 'timmy', 'helena']
```

```
[60]: l[1:3]
```

```
[60]: ['john', 'timmy']
```

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

Lists

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

```
[61]: l[2:]
```

```
[61]: ['timmy', 'helena']
```

```
[62]: l[:2]
```

```
[62]: [1.2, 'john']
```

Lists

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

```
[63]: l
```

```
[63]: [1.2, 'john', 'timmy', 'helena']
```

```
[64]: l[::2]
```

```
[64]: [1.2, 'timmy']
```

```
[65]: l[2::]
```

```
[65]: ['timmy', 'helena']
```

Lists

  • Use [from:to:step] to reverse a list:

```
[66]: l[::-1]
```

```
[66]: ['helena', 'timmy', 'john', 1.2]
```

Lists

  • Use * and + to repeat and join lists:

```
[67]: l*2
```

```
[67]: [1.2, 'john', 'timmy', 'helena', 1.2, 'john', 'timmy', 'helena']
```

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

```
[68]: [1.2, 'john', 'timmy', 'helena', 'helena', 'timmy', 'john', 1.2]
```

Lists

- Lists are mutable

```
[69]: l
```

```
[69]: [1.2, 'john', 'timmy', 'helena']
```

```
[70]: l[0] = "thomas"
```

```
[71]: l
```

```
[71]: ['thomas', 'john', 'timmy', 'helena']
```

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

```
[73]: l.append([4,3])
```

```
[74]: l
```

```
[74]: ['thomas', 'john', 'hannah', 'laura', [4, 3]]
```

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

List methods

Source: Python for Finance, 2nd ed.

Dictionaries

- Dictionaries (`dict`) store and retrieve data associated with keys.
- They are so-called **key-value stores**. A `dict` entry is a `key:value` pair.
- Keys within a dictionary must be unique.
- Dictionaries are mutable and can contain heterogeneous data.

Example

```
[75]: d = {
          'Name': 'Angela Merkel',
          'Country': 'Germany',
          'Profession': 'Chancelor',
          'Age': 64
      }
      type(d)
```

```
[75]: dict
```

```
[76]: print(d['Name'], d['Age'])
```

```
Angela Merkel 64
```

```
[77]: d.keys()
```

```
[77]: dict_keys(['Name', 'Country', 'Profession', 'Age'])
```

```
[78]: d.values()
```

```
[78]: dict_values(['Angela Merkel', 'Germany', 'Chancelor', 64])
```

```
[79]: d.items()
```

```
[79]: dict_items([('Name', 'Angela Merkel'), ('Country', 'Germany'), ('Profession',
      'Chancelor'), ('Age', 64)])
```

Example (cont'd)

```
[80]: for item in d.items():
          print(item)
```

```
('Name', 'Angela Merkel')
('Country', 'Germany')
('Profession', 'Chancelor')
('Age', 64)
```

```
[81]: for value in d.values():
          print(type(value))
```

```
<class 'str'>
<class 'str'>
<class 'str'>
<class 'int'>
```

11

Method	Arguments	Returns/result
d[k]	[k]	Item of d with key k
d[k] = x	[k]	Sets item key k to x
del d[k]	[k]	Deletes item with key k
clear	()	Removes all items
copy	()	Makes a copy
has_key	(k)	True if k is a key
items	()	Iterator over all items
keys	()	Iterator over all keys
values	()	Iterator over all values
popitem	(k)	Returns and removes item with key k
update	([e])	Updates items with items from e

Dict methods

Dict methods

Source: Python for Finance, 2nd ed.

Sets

- A set is an unordered collection of objects and does not have duplicates.

```
[82]: participants = set(['Tom', 'Julia', 'Tom', 'Lily', 'Julia', 'Tom'])
```

```
[83]: participants
```

```
[83]: {'Julia', 'Lily', 'Tom'}
```

```
[84]: letters = set(['a', 'a', 'b', 'c', 'c', 'c', 'd', 'd'])
```

```
[85]: letters
```

```
[85]: {'a', 'b', 'c', 'd'}
```

Sets

- Sets support mathematical operations as in mathematical set theory.

```
[86]: s = set(['a', 'b', 'c', 'd'])
      t = set(['b', 'c', 'd', 'e'])
```

```
[87]: s.union(t)
```

```
[87]: {'a', 'b', 'c', 'd', 'e'}
```

```
[88]: s.intersection(t)
```

```
[88]: {'b', 'c', 'd'}
```

```
[89]: s.difference(t)
```

```
[89]: {'a'}
```

```
[90]: t.difference(s)
```

```
[90]: {'e'}
```

0.5 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:

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

```
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:

```
[92]: 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))
```

```
100 > 4
```

Control flow: `for`-loops

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

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

13

```
John is taking this lecture.
Alina is taking this lecture.
Timmy is taking this lecture.
Helena is taking this lecture.
```

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

```
[94]: for (i) in range(4):
          print(i)
```

```
0
1
2
3
```

Control flow: `while`-statement

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

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

```
0 is smaller than 4.
1 is smaller than 4.
2 is smaller than 4.
3 is smaller than 4.
```

- The `break` keyword interrupts control flow:

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

```
0 is smaller than 3
1 is smaller than 3
2 is smaller than 3
```

0.6   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.

```
[97]: def circle_circumference(radius):
          return 2 * 3.14159 * radius * 2
```

```
[98]: circle_circumference(5)
```

```
[98]: 62.8318
```

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:

```
[99]: 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)
```

```
23
[99, 42]
[99]
77
[99, 42]
[28]
```

Function syntax

- Function arguments can be given default values:

```
[100]: def plus10(x = 50):
           return x + 10
```

```
[101]: plus10()
```

```
[101]: 60
```

```
[102]: plus10(20)
```

```
[102]: 30
```