Optiver

Optibook Python Reference

2025

Table of Contents

Introduction

This document provides an introduction to Python programming for interacting with the Optibook virtual exchange. It is targeted at participants with some experience in another language/other languages, but still at a beginning level, and inexperienced with Python. This text aims to teach only what is required for understanding interactions with Optibook and contains simplifications and omissions where didactically preferable, as such it is not a full language reference.

In large, the document follows along with the Informal Introduction to Python, which is part of the official tutorial hosted on the Python website. Both that informal introduction chapter and the full tutorial are excellent resources. If time allows, in particular chapters 3 through 6 of the tutorial are worth going through for a more robust introduction to the language.

If you are already familiar with Python, section 3 contains details on the Optibook client.

1 Programming Environment

1.1 Python

Python is an interpreted programming language. This means that a Python program is defined in a text file, which is read and executed by a Python interpreter executable file, e.g. *'python.exe'* on Windows or *'python.so'* on Linux. To write a Python program is to write a text file which contains the sequence of commands to be ran, these files are stored with the extension *.py,* as in *'my_program.py'*. This file is not compiled but read and ran directly.

Multiple versions of the Python language have been defined over the years, the latest at time of writing being 3.9.6. There are large differences between major versions, such as between 2.7 and 3.9, but the differences between minor versions, say. 3.7.11 and 3.9.6, are small, especially when sticking to the more basic language features. As such this guide should apply equally to all sub-versions of Python 3.

1.2 Code Editors

It is possible to use any text editor to write and edit text files containing Python code. Something as simple as Windows notepad can work. The most common choice however is to use a dedicated code editor called an Integrated Development Environment, or IDE for short. IDEs offer a lot more than only text editing features, they will usually have some form of automatic code completion, a small file browser, side by side display of different files, the ability to run programs directly from within the editor, debugging features, etc. Many different IDEs exist, with two well-known ones being PyCharm and Spyder.

To avoid any differences between setups, we have opted for participants of an Optibook programme to write and run their code through the cloud-based service hosted on Amazon Web

Services (AWS). What this provides is an environment that is correctly setup from the first time you launch it, offering a pre-installed version of Python, a browser-based IDE and the ability for us to preload the Optibook libraries and any example code files.
More details about logging in to the Cloud9 environment should be provided to you via extra materials or lectures in your programme.

2 Python Language Guide

2.1 How to read this guide

When following this guide, it is highly recommended that you test and modify the code examples given as you read them. Programming is very much an applied skill; a good analogy is wood-working. There are infinitely many ways to make something work, but mastering the craft is knowing what to do, and importantly what not to do, each step of the way. As with wood-working, you learn programming by doing, not by reading. Take the examples given as interesting areas to explore rather than as facts to be memorized.

2.2 Comments, Printing and Variables

All lines of a Python file are interpreted to be ran as code. The exception are **comments**, which are any characters that follow a # sign.

```
# this is a comment
a = 2 * 3 # this is an in-line comment
```

As it will be used throughout the other examples, let’s first cover the built-in **print function**. This function allows us to display the contents of any variable, or whatever input we provide to it, to the screen. That makes it an extremely useful debugging tool. The following shows some examples:

```
print('test') # prints test
a = 2
print(a)     # prints 2
b = 4
print(a, b)  # prints 2, 4
```

Variables are defined by writing a variable name of your choice followed by an equals-sign, as for ‘*a*’ and ‘*b*’ in the example above, which store the values 2 and 4. The contents of this variable can then be re-used further on in the code.

When using longer variable names it is the preferred style to join words together with the “_” sign, as in *‘this_is_a_long_variable_name’*, rather than using camel casing as in *‘thisIsALongVariableName’*, but the interpreter allows any combination of characters, non-capitals and capitals alike.

The language is case sensitive, so *'AVar'* is an entirely different variable from *'avar'*.

2.3 Basic Data Types

Numbers are represented as **integer** or **float**. Both number types work as you expect, one can apply usual arithmetic on them: +, -, /, * for addition, subtraction, division and multiplication respectively, and ** for powers. If two integers are used in a calculation and it would result in a non-integer number, the conversion to float is automatic.

Integers are exact, but only store whole numbers. Floats are inexact, but can store digits after the decimal point. They are inexact in the sense that a number like 9.99999999 could be rounded to 10.0 or vice versa, the rounding happens according to rules that are not in the scope of this guide, but is typically very minor for numbers with less than 12 digits before and 6 digits after the decimal point.

To define an integer in code, write out a number without a decimal point, to define a float, include a decimal point, see the following examples:

```
a = 17   # int
b = 6    # int
c = a / b # result stored in c is a float

d = 5.0   # float
e = 4.    # also float
```

A **string** is the data type that is used to store text, strings are defined with quote-signs (' or "). A few different combinations of quote-signs can be used, all equally defining a string. This is useful as displayed by the following examples:

```
# simple strings
a_string = 'Test one two..'
another_string = "Three four five."

# multi-line strings can be defined by using triplets of
# quote-signs, they may also be used single-line
multi_line_string = '''First line,
Second line,
Third line.'''

single_line_triplet = '''This is a normal string.'''

double_quotes_triplet = """Triplets work with
double-quote signs too."""
```

```
# embedding quote signs in strings
quote_in_string = "Single-quote sign inside a string: '. Nice."
```

Strings can be added together to form a new string with an addition sign.

```
a = 'Hello '
b = 'world'
c = a + b       # 'Hello world'
d = a + a       # 'Hello Hello '
```

There is a very useful function for string formatting which we use extensively in the algorithm examples. It is called **f-string formatting** and the notation is as follows:

```
name = 'Person'
print(f'Hi, {name}!') # prints 'Hi, Person!'
```

What is going on here is that for a string that is prepended with an *f*, any text that is located between curly brackets, even inside of the string, is evaluated as Python code first, with the result placed back into the string. This saves a lot of duplicated typing when logging information to the screen. It can take any type of input, e.g. numbers and lists too. The notation might seem odd, but it starts to make more sense when you learn about other, out-of-scope, string-related features. For now, just take it on faith.

A **boolean** can take on only one of two values, *True* and *False* (note the capital first letter). It is a useful type to store the results of comparisons and they can be manipulated using Boolean logic (see section 2.7.1).

```
a = True         # True
b = False        # False
c = not False    # True
d = not a        # False
e = a or b       # True
```

The none-type is a special type that represents the absence of a value. E.g. later, when working with functions, If a function does not explicitly return a value it returns **None** instead.

```
val = None      # A None-type
```

2.4 Compound Data Types

Python knows several compound data types, used to group together other values. The most common is the **list**, which can be written as a list of comma-separated values (items) between square brackets. Contrary to the perhaps familiar array-type, often used in other languages as the basic way to define a sequence, lists may mix items of different types. Usually however, the items all have the same type, as in the example below.

```
squares = [1, 4, 9, 16, 25]
```

One can access contents of a list with **indexing**, the first element has index 0.

```
squares = [1, 4, 9, 16, 25]
print(squares[0])   # prints 1
print(squares[2])   # prints 9
print(squares[-1])  # prints last element, 25
```

It is also possible to subset the list with **slicing**:

```
squares = [1, 4, 9, 16, 25]
print(squares[0:2]) # prints [1, 4]
print(squares[1:3]) # prints [4, 9]
print(squares[:3])  # prints [1, 4, 9]
print(squares[2:])  # prints [9, 16, 25]
```

And we can change a particular item to another value:

```
values = [1, 3, 5]
values[2] = 6  # After this line values gets set to [1, 3, 6]
```

Another important compound data type is the **dictionary**, it stores key-value pairs. Where lists store ordered data, dictionaries are unordered by definition. They are efficient at storing mappings from one value to another, for access in unknown order. A dictionary is defined with curly brackets as in the following example:

```
currencies = {'USA': 'USD', 'Netherlands': 'EUR', 'Germany': 'EUR', 'Japan':
'JPY'}

currencies['China'] = 'CNY'   # Add Chinese Yuan

print(currencies['Japan'])  # Will print 'JPY'
```

In the example above we mapped countries (string) to currencies (string), but we can map many types and are even allowed to mix different types together.

All existing data types can be used as key and value in a dictionary, with the single demand that the key is a so-called immutable type. That means, we cannot use a list as a dictionary's key. Intuitively, a list's contents can change under the hood, which the dictionary would not handle well, so it is forbidden.

We can obtain all keys of the dictionary with the *.keys()*-method (more on methods in section 2.9). This is useful for looping all entries (more in section 2.7.4).

```
currencies = {'USA': 'USD', 'Netherlands': 'EUR', 'Germany': 'EUR', 'Japan':
'JPY'}

# Prints USA, USD. Netherlands, EUR. Etc. Etc.
for country in currencies.keys():
   print(country)
      print (currencies [country])
```

2.5 Operators

To add two numbers together, be they float or int, one sensibly uses the + sign, as in the notation of normal arithmetic. At the same time, when adding two strings together, we can also use the + sign and the behaviour is quite different, it concatenates the inputs. On lists, the behaviour is a concatenated list, different yet again. Consider these examples:

```
a = 1
b = 2
print(a + b)  # 3

c = '1'
d = '2'
print(c + d)  # '12'

e = [1]
f = [2]
print(e + f)  # [1, 2]
```

What is going on? The + sign has a special meaning, it is an example of a so-called **operator**. Operators work on inputs, and how they act is dependent on the type of those inputs.
In Python, the fact that addition is possible on strings is not a feature of the + sign, but rather a feature of strings, defining *how addition should be applied to them*. And that holds for all operators: they can be used on many different object types, and might have different effects depending on the types. On some types, operations might not be defined, e.g. attempting to apply the division operator / with a string as the divisor will result in an error.

Examples of operators are:

- The assignment operator: =
- Arithmetic operators: +, -, /, //, *, **, %
- Comparison operators: ==, <=, <, >, >=, !=
- Bitwise operators: ~, &, |, <<, >>, ^
- Boolean operators: not, and, or
- Identity operators: is, is not
- Membership operators: in, not in

Most of the operators work on two inputs, as with the + sign, the left-hand side and the right-hand side. Some of them also work on a single input, for example: *not*, bitwise not ~, and the simple negation sign -. That input is then provided on the right-hand side.

It would be convenient to think of the arithmetic operators as always applying arithmetic to numbers, but as you now know, that's not quite true. They apply arithmetic when it pertains to ints and floats, but can and do apply other operations to strings, lists, sets, dictionaries and other (custom) types.

It is not important to memorize all operators and how exactly they act on all different types. Rather you should understand that the meaning of operators depends on the types they are applied to, and know for the specific operations you do apply to your own variables, how they will act.
Most operations implemented by operators are very intuitive, but not always. As an example, *a* ^ *b*, where *a* and *b* are integers, does not actually raise *a* to the power of *b*, as is common notation on some calculators. To do that we should use *a* ** *b*. Can you figure out what *a* ^ *b* does instead? The lesson is this: If you are not sure what to expect on a specific operation, look it up first.

As a final remark on operators: It is crucially important to distinct the the assignment operator = from the comparison operator ==. While the notations look similar, they are used for completely different purposes. The assignment operator = is a very core operation which sets a variable on the left-hand side to a specific value provided on the right-hand side, whereas the comparison operator == compares two variables for equality (the exact method of which depends on the types specified). They have nothing to do with each other!

2.5.1 Assignment operators

A variable is assigned a value, or pointed to an object, through the assignment operator =. For convenience, there also exist assignment operators which combine arithmetic operations with assignments. The following operators all exist: +=, -=, *=, /=, //= **=, %=.

The following example demonstrates the behaviour of the += assignment operator. The other assignment operators behave similarly: first the arithmetic operation is applied, followed by the assignment operation.

```python
a = 5
b = 2

a += 3  # Equivalent to a = (a + 3)
print(a)  # 8
```

```
a += b  # Equivalent to a = (a + b)
print(a) # 10
print(b) # 2, variable b is not affected
```

2.6 Objects

So far we have considered variables with contents exclusively containing data. Not all data is of the same kind, so we distinct different types: numerical with *int* and *float*, textual with *string*, or combining different data points, compounding them with *list*s and *dict*s.

We reference that data with variables, and those variables point to a location in computer memory where its contents are stored. The things in memory which are being pointed to by the variables are called **objects**. So we have integer objects, float objects, string objects, list objects, etc.

2.6.1 Non-data Objects

Not all objects have to be data. In more advanced programming, an object might represent an abstraction around something entirely different, such as an internet connection, a piece of hardware like a printer or a file on the hard disk.

All of the following lines of code would be unsurprising to find within a Python program, if first set up correctly using the right packages of course:

```
printer = Printer()
printer.print_page('Test Page 123')

http_req = HTTPRequest('www.google.com')
response = http_req.get()
print(response.content)

file = open('test.txt')
print(file.read_line())
```

We can still interact with these objects, manipulate them and pass them around as with objects which contain data. In Python, they are the same kind of thing, an object.

An object which is not data that will be used when interacting with Optibook is the Exchange object. It is detailed fully in the final chapter, but it is an abstraction of the Exchange connection, like the examples above were abstractions as well. It implements methods such as *exchange.insert_order(...)* and *exchange.get_pnl()*. This special object type is what will finally allow us to implement a trading algorithm to execute trades from Python.

2.6.2 Objects vs Variables

Why do we go through so much effort to distinct the term object from the term variable? Don't they both serve the same purpose, to contain some data or other type of unit to interact with?

Almost. It is easy to mix the two terms up, but the above sentence only describes objects. Objects store the data and the methods to interact with them. Variables point to these objects.

Due to this difference, it is possible to do things which might otherwise seem unintuitive. For example, we can have multiple variables referencing the same object, and when one "variable is changed", so is the other. (Actually, only the referenced object changes.)

This becomes very clear when considering mutable objects such as lists. **Mutable objects** are those objects which allow dynamically changing their contents. Lists are mutable as we may change/overwrite what each element contains, without redefining the whole list.

Let's see an example.

```
numbers = [1, 2, 3, 4]
values = numbers

values[0] = 10

print(numbers)  # prints [10, 2, 3, 4]
print(values)   # prints [10, 2, 3, 4]
```

The change to *values* on the third line was also applied to the variable *numbers*! The example is trivial to understand if the distinction between variables and objects is clear, but otherwise it could be quite confusing. The reason for this is of course that there exists only one object, the list initially containing [1, 2, 3, 4] and later containing [10, 2, 3, 4]. On the first line we point *numbers* to that new object, and on the second line we point *values* to that same object. After they reference the same object, on further occurrences in the code, we could as well swap around the variable *numbers* with the variable *values* anywhere we like, resulting in the same outputs, because they are pointing to the same list.

This example is perhaps still relatively straight-forward. But in later contexts, where variables are being passed around through functions and the like, the only way to understand what's truly going on is to follow along with where the objects are going. That is why it is important to keep a conceptual distinction between the two terms.

2.6.3 Mutable vs Immutable objects

The example with the modified list was possible because the list is mutable, we can change its contents, what is inside of it. Immutable objects don't allow this type of modifications.

Examples of immutable objects are ints and floats. How can we change the contents of a 5? We can't, it doesn't have contents, it simply is 5. All we can do is point a variable to a 6 instead, but that hasn't changed the contents of 5, it has simply pointed the variable elsewhere. Strings are also examples of an immutable type, hence this works fine:

```
a = 'Test one two..'
print(a[5:8])     # Prints 'one'
```

but these throw an error:

```
a = 'Test one two..'
a[5:8] = 'two'     # Throws error, cannot modify
a[0] = 't'         # Throws error, cannot modify
```

Which object types are mutable vs immutable are well thought-out features of the language and will become natural with experience. One place you will find a constraint on mutability is in the keys of a dictionary, which need to be immutable. One can understand this intuitively, if dictionaries are like look-up tables, if the contents of the keys of the look-up table change under the hood, the look-up table loses its consistency.

2.7 Control Flow

2.7.1 If Statements

An **if-elif-else statement** allows to conditionally execute code. Which lines of code exactly are conditional on the preceding if-statement, are together called a code block.

A **code block** is defined through the indentation-level of the line of code. This is an uncommon feature of Python, many other languages opt for defining a code block using either curly brackets or by including an explicit END IF.

The following example shows how to write an if-elif-else statement in Python. Each print-statement is it's own, single-line code block. The two print-statements under *else:* are in the same code block, so will both run only if the else-condition is met, i.e. all other conditions are False. The very last print statement will always run as it falls outside of the if-statement altogether.

```
# Because val = 6, this prints both 'The value was 5 or higher' and
'This is also within the else-code block'

val = 6
if val == 1:
  print('The value was one.')
elif val < 1:
  print('The value was less than one.')
elif val >= 2 and val <= 4:
  print('The value was between two and four.')
else:
  print('The value was 5 or higher.')
  print('This is also within the else-code block.')

print('This falls outside of everything and is always printed.')
```

Besides the if-statement you can also see the **comparison operators** in action in the above example.

Also on display in the example are the **Boolean operators**, *and* and *or*. These compare Boolean values which are True or False in the expected way. When combined with round brackets for grouping, and the **not-keyword** for negation, any desired comparisons can be made. Whenever in doubt of what the comparison order could be, use more brackets.

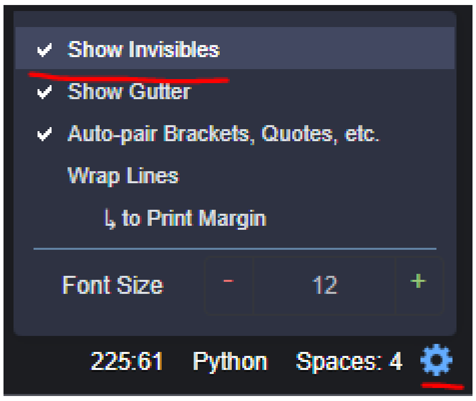
```
a = 2
b = 3
if (a == 2) or not (a > 2 and b <= 3):
    print('The condition was True.')
```

2.7.2 Indentation

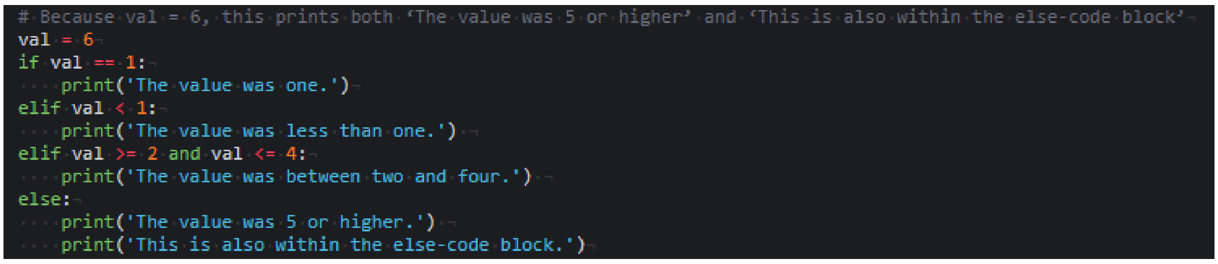
The indentation used to define a code block is normally four space characters. Theoretically, it is also allowed to be a different amount of space characters or to be created with tabs instead of spaces. The strict rule being that there have to be a consistent amount and type for each line within the block. While any indentation choice is allowed, it is a strong standard that what should be used is four spaces. It simply avoids a lot of hassle not to deviate. For nested blocks use eight spaces, deeper nesting twelve spaces, and so on.

People usually quickly tire of pushing the spacebar key four times for a single indent. For that reason, many of the IDEs, including Cloud9, place 4 space characters whenever the tab key is pressed, instead of a single tab character. This is called a **soft-tab**. Because of this feature of replacing tabs with spaces, it does not matter whether you indent by pressing spacebar 4 times, or by pressing tab once, the result in the text-file is the same, 4 spaces added.

To further confuse the novice programmer (and speed up jumping around the code), when moving the text-cursor over an indent of four spaces with the arrow keys, in Cloud9, the editor jumps over all 4 spaces as if it was indeed a single tab character. To avoid too much confusion, it is helpful to enable the *Show Invisible (Character)s* setting, found in the bottom-right of a screen when a .py file is open in the editor:



This shows what's truly going by adding gray dots for spaces:



```
# Because val = 6, this prints both 'The value was 5 or higher' and 'This is also within the else-code block'
val = 6
if val == 1:
    print('The value was one.')
elif val < 1:
    print('The value was less than one.')
elif val >= 2 and val <= 4:
    print('The value was between two and four.')
else:
    print('The value was 5 or higher.')
    print('This is also within the else-code block.')
```

2.7.3 Thrutiness

An if-statement runs its corresponding code block if what is supplied to it evaluates to the Boolean value True and the else-block if it evaluates as False. But this is not the complete picture. We can pass objects of non-Boolean type into the if-statement, and depending on the type's definition, it has a so-called '**truthiness**'. That's a bit oddly named, but a useful concept nonetheless. Here are a few examples which have Falsiness, e.g. they behave like False in an if-statement:

```
if None:
    print('Truthy.')  # This will not run

empty_list = []
if empty_list:
    print('Truthy.')  # This will not run

if 0:
    print('Truthy.')  # This will not run
```

The following are truthy:

```
filled_list = [0, 1, 2]
if filled_list:
    print('Truthy.')  # This will run

if 1:
    print('Truthy.')   # This will run
```

Imagine a function *load_birthdays()* which loads the names of people with birthdays from a database, and another function *greet(people),* which prints greetings to each of the people in the people-list. Due to the way truthiness works, we can then write the following, quite subtle, code:

```
birthdays = load_birthdays()
if birthdays:
    greet(birthdays)
else:
    print('Nobody had their birthday today.')
```

2.7.4 Loops

A **while-loop** runs as long as the condition supplied to it evaluates as True. E.g.

```
# This prints 0, 1, 2, 3 and 4 in sequence and then stops
val = 0
while val < 5:
    print(val)
    val = val + 1
```

A **for-loop** is defined slightly differently in Python than in many other languages. In many languages for-loops count over numbers, Python for-loops are more general and instead iterate (loop) over elements of a sequence. That looks as follows:

```
squares = [1, 4, 9, 16, 25]
for val in squares:
    print(val)
```

The print-statement in the code block below the '*for <x> in <y>:*'-line is ran for every element in the list of squares, therefor that print-line runs 5 times. First, *'val'* will take on the value 1, get printed, then take on the value 4, get printed, take on the value 9, get printed, and so on. We can choose how to name the current element (*'val'* in the example) freely, so the following has the exact same result:

```
squares = [1, 4, 9, 16, 25]
for square in squares:
    print(square)
```

2.8 Functions

2.8.1 Defining a Function

Functions are used to group a set of related operations together for re-use, to avoid code duplication and improve code clarity. A function is defined using the def keyword as in the example below. They can take zero, one or more input parameters called **arguments**.

Here the function name is *count_and_print.* The name is chosen freely to describe its behavior. This particular function takes two arguments, *start* and *end,* the variable names are also freely chosen. For example, we could have just as easily chosen *n* and *m*, being mindful to replace all instances of these variables in the function body as well.
A **docstring** can be included after the line with the def keyword, name and arguments. It is written between a triplet of double quotes. This docstring is for the purposes of documenting to others what the function does, but has no special meaning code-wise.

Finally, we have the function body, which is all the code in the code block below the def-line, and is what is ran when the function is called.

```python
def count_and_print(start, end):
    """
    This function consecutively counts up from the value <start>
    to the value <end> and prints all values in between.
    """
    val = start
    while val < end:
        print(val)
        val = val + 1
```

Using a function elsewhere in code is the reason it exists, and is called **calling a function**. The calling of a function can be thought of as the interpreter jumping into the function body and running that code block line by line, appropriately parameterized based on the supplied arguments, before jumping back into the main body of code.

Functions can also be called from within other functions, and in that way we can get a whole stack of function calls going, each returning back to the point they left off when finished.
Each of the following lines calls the above function with different inputs.

```python
count_and_print(2, 5)  # Prints 2, 3, 4
count_and_print(8, 10) # Prints 8, 9
count_and_print(1, 1)  # Prints nothing
```

A function can include a **return statement** to provide output, but it does not have to. Functions that do not, such as the above *count_and_print*, simply return the special value *None*.

Some beginners get confused between the action of printing and returning a value. They are very different operations. A print-statement displays whatever its input to the *screen* for human consumption, whereas a return-statement provides the returned value for further use in the *code*.

```
ret_value = count_and_print(2, 5)  # Prints 2, 3, 4
print(ret_value)                   # Prints None
```

Here is an example of a function that does return a value:

```
def add(a, b):
    """
    Adds together the inputs and returns the result.
    """
    c = a + b
    return c

s = add(2, 5)  # Prints nothing
print(s)       # Prints 7

print(c)       # Gives an error-message, c not defined (outside
               # of the function)
```

It is very important to understand the difference between *c* and *s* in this example. Inside of the function, the summed value of *a* and *b* is stored in a variable called *c*, it is only available from within the function body. After the value of this *c* is returned, it is stored in *s* externally. It is never known as *c* outside of the function body, only the value 7 is returned and by writing '*s = …*' we denote that we want to store that value in a new variable *s*.

Again, the names *c* and *s* have been chosen arbitrarily, we could replace every instance of *c* by *foo* and *s* by *bar* and the functionality would be exactly the same.

Besides providing output, a return statement also ends operation of the function, regardless of where it is placed. If all that is desired is to stop further operation of the function, returning nothing is a common way to do so. The below shows an example, it will not trigger a ZeroDivisionError, because it returns before it would attempt to do such a division.

```
def divide(a, b):
    """
    Divides a by b, if b is zero, prints a message and quits.
    """
    if b == 0:
    print('Attempting division by zero, returning nothing.')
        return
    c = a / b
    return c
```

2.8.2 Default Arguments

It is possible to supply default arguments in a function definition. Then, if those arguments are not provided in a function call, they are filled in to be the default. To make consistency in the interpretation of the function call more straight-forward, those arguments with default values are only allowed to come after arguments without default values.

```
def add(a, b=10, c=20):
    """
    Adds together the inputs and returns the result.
    """
    d = a + b + c
    return d

s = add(1)
print(s)    # Prints 1 + 10 + 20 = 31

s = add(1, 5)
print(s)    # Prints 1 + 5 + 20 = 26

s = add(1, 5, 10)
print(s)    # Prints 1 + 5 + 10 = 16
```

2.8.3 Keyword Arguments

If a function has many arguments, it can be difficult to keep track of the order and which argument is which. Then it's possible to supply those arguments by name, called **keyword arguments**. In contrast, the arguments that were resolved based on position are called **positional arguments**.

Using keyword arguments can make tractability better. It is also allowed to add line breaks in a function call as in the example below, making for even easier reading.

```
def greet(name, country, number_of_times):
    """
    Prints greeting to <name> from <country> to the
    screen <number_of_times> times.
    """
  count = 0
    while count < number_of_times:
    count = count + 1
    print('Hi' + name + ', from' + country + '!')
```

```
# Here we call it without keyword arguments
greet('Student',
      'The Netherlands',
      5)

# But the following, using keyword arguments, is easier to follow
greet(name='Person',
      country='Germany',
      number_of_times=10)

# When using keyword arguments, we may choose a random order
greet(country='France',
      number_of_times=8,
      name='Citizen')

# We can combine positional and keyword arguments, first
# resolving from left to right, and then based on keyword
greet('People',
      number_of_times=3,
      country='Belgium')
```

Don’t confuse the notation for keyword arguments with default arguments. They look similar, but default arguments exist inline with the function definition, whereas the provided keyword arguments are written on the line where the function call is made.

2.8.4 Keyword-Only Arguments

The designer of a function may opt to enforce passing of arguments by keyword rather than by position, if it seems more appropriate. This is done by adding an asterisk before all arguments that are keyword-only, as in the following:

```
def greet(name, *, country, number_of_times):
  count = 0
    while count < number_of_times:
    count = count + 1
    print('Hi' + name + ', from' + country + '!')
```

Attempting to pass positional arguments for *country* or *number_of_times* now results in an error:

```
# This is fine
greet(name='Person',
      country='Germany',
      number_of_times=10)
```

```
# This too because the <name> argument is before the asterisk
greet('People',
      number_of_times=3,
      country='Belgium')

# But this results in an error
greet('Student',
      'The Netherlands',
      5)
```

2.9 Object Methods & Attributes

The object types we have covered so far allow for several different interactions with the underlying object, besides being simple stores of values. To list a few of many: the String type implements the functionality to return a capitalized copy of itself, the list type allows appending a value to the end of it or removing a value from any position, and the dictionary allows access to a sequence of all of its keys.

How do we use these functionalities? With so-called object **methods**. Methods are functions defined on an object, and used with the syntax '*variable_name.method_name(arguments)*'.

```
a_string = 'test one two..'
print(a_string.capitalize())  # 'Test one two..'

squares = [1, 4, 9, 16, 25]
squares.append(36)
print(squares)              # [1, 4, 9, 16, 25, 36]

val = squares.pop(2)
print(val)                  # 9
print(squares)              # [1, 4, 16, 25, 36]

currencies = {'USA': 'USD', 'Netherlands': 'EUR'}
print(currencies.keys())        # dict_keys(['USA', 'Netherlands'])
```

The available methods depend on the object type. It would of course be non-sensical to capitalize a list or get the keys of a string.

Some methods modify the internal state of an object, such as the *.pop()* and *.append()* methods above, but some of them do not, such as the *.capitalize()* and *.keys()* methods, which return something based on the state instead. Some of them do return a value, such as *.pop()*, *.capitalize()* and *.keys()* while some others don't, such as *.append()*.

Here we see that the *.append()* method modifies the list it is used on, but it does not return any value. Try to follow along with exactly what happens here, misunderstanding this is a common cause of error.

```
squares = [1, 4, 9, 16, 25]
val = squares.append(36)
print(val)            # None
print(squares)        # [1, 4, 9, 16, 25, 36]
```

Attributes are similar to methods, but they are additional variables that are attached to an object, rather than functions. Where a method is called with a statement like *'obj.method(args)'* an attribute is not called but just used directly, as in '*obj.attribute*'.

Most of the basic types we have covered do not have any attributes, only methods. In using the Optibook Client package, we will use attributes often though, for example in object types such as a *TradeTick* which has attributes like *.price*, *.volume*, *.timestamp*, etc.

To provide an example with a type we've already covered, floats do have two attributes, *.real* and *.imag*. These attributes allow specification of complex numbers (in the mathematical sense), the below example displays how that works.

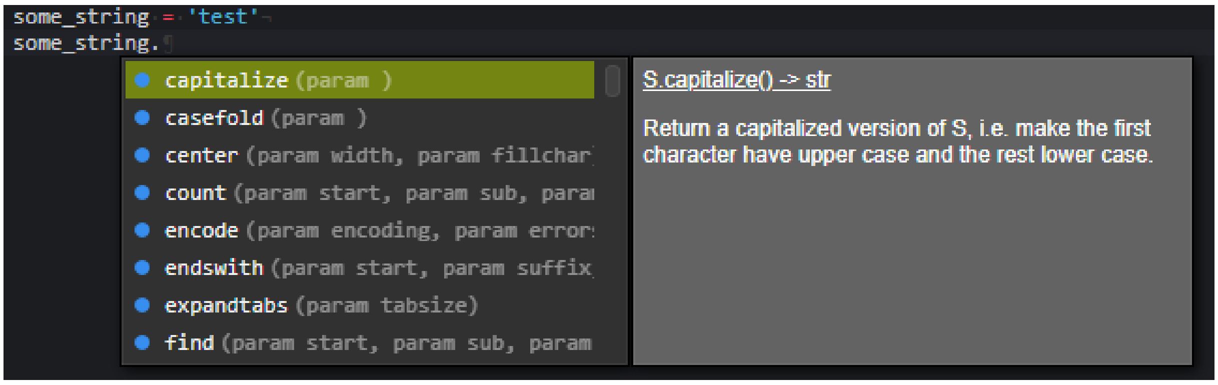
```
val = 1.0            # Real number
print(val.real)    # 1.0
print(val.imag)    # 0.0

val = 1.0 + 2.0j # Complex number, defined in Python via j
print(val.real)  # 1.0
print(val.imag)  # 2.0

val = val * 2.0
print(val.real)  # 2.0
print(val.imag)  # 4.0
```

These attributes are not writable (settable), only readable (gettable). Trying *'val.real = 2.0'* will result in an exception being raised. Know that this is not always the case, it depends on the specification of the attribute.

2.9.1 Auto-complete

How do you find which attributes and methods are available on a type? There's usually documentation, e.g. the Python website has the full specifications of the types we've covered so far, and for the Optibook Client we've provided online docs as well.

There's also a more practical way, which is through the IDE. With its **auto-complete** functionality, if you type a variable name followed by a period and the IDE is able to figure out the object type, it displays a pop-up with a long list of all available attributes and methods. By using the up and down arrow-keys, you get a quick view of the required arguments, return values and docstrings. With some educated guessing of what *might* be available, you should be able to quickly find what you need.



```
some_string = 'test'
some_string.
```

```
capitalize (param )
casefold (param )
center (param width, param fillchar
count (param start, param sub, para
encode (param encoding, param error
endswith (param start, param suffix
expandtabs (param tabsize)
find (param start, param sub, param
```

S.capitalize() -> str

Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.

2.9.2 Type Hints

It should be clear by now that Python is a so-called untyped language. In typed languages, when a new variable is created or defined as a function argument, the type need to be specified, and storing anything else in that variable will result in errors. Not so in Python.

The idea in an untyped language is that functions and other code are written in such a generic way, that the type becomes irrelevant to supply. If we have a function *add* then it should hardly matter whether the inputs are two floats, two ints, a float and an int, or an int and a float. In all cases the function body should read something like '*return a + b*'. Even if we pass two strings into this function it would work, albeit concatenating instead of adding, but still not acting in a completely unexpected way.

What happens if we pass in something unaddable, like a dictionary? An error is raised as the + operator is not defined for it. The philosophy is to try and fail if needed, rather than to check and prevent any failures.

This untypedness of the language certainly has its advantages and proponents, but is has also proven problematic in a few areas. For one, code tractability is lower. Also importantly, an IDE/code editor has trouble following along. If it is not clear what type an argument or variable is, auto-complete cannot function.

For this reason **type-hints** were introduced into the language. They are not proper types, as it is perfectly allowable to provide an argument of a different type into a function with a type-hint without running into any complaints or exceptions. However, the type hints do *hint* to the IDE and the reader what the expected type is, such that auto-complete can work and we gain some of the benefits of a typed language back. The notation is given below:

```
def greet(name: str, number_of_times: int) -> str:
  greeting = 'Hi ' + name '!'
  print(greeting)
```

```
    return greeting

the_greeting = greet('Person', 5)
```

The two argument types are string (str) and integer (int), and the return type is a string (str). When entering *'name.'* inside of the function body, Cloud9 auto-complete will show all methods of a string type. When writing *'number_of_times.',* those of an int type. And the same is true when writing '*the_greeting.*' in the main code body below the function call. This is all very useful!

For compound types, type-hinting is unfortunately a bit more complex, as the dedicated type-hint type needs to be imported before use and the notation starts to deviate a little from common Python. It shows that the feature was not added into the initial design of Python but was rather patched on later.

The following shows an example. While it is useful to be able to read this, in small projects it is not needed to replicate it in your own code.

```
from typing import List, Dict

def func(a: List[int], b: Dict[str, float]) -> None:
    print(a, b)
```

The *a* argument is expected to be a list of integers. The *b* argument is expected to be a dictionary with string-keys mapping to float-values. We can now get auto-complete functioning even on something like *'a[2].'* (int) and *'b["Germany"].'* (float).

2.10 Exceptions (Errors)

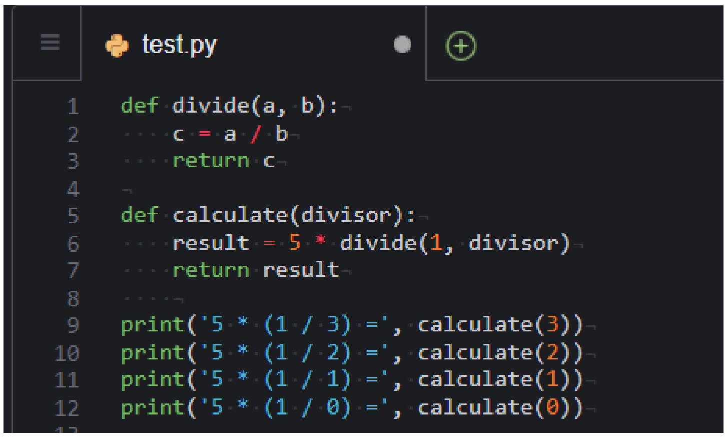
Exceptions are error messages which are displayed if something unexpected or undefined happens. When not handled in the code automatically (out of scope of the tutorial), the result of an exception being thrown is that the code is halted on the line where it happened, and no further code is executed. There are many reasons an exception might be thrown, below are a few examples:

```
print(n)   # use the variable n which we have not defined
NameError: name 'n' is not defined
```

```
1 / 0      # divide by 0
ZeroDivisionError: integer division or modulo by zero
```

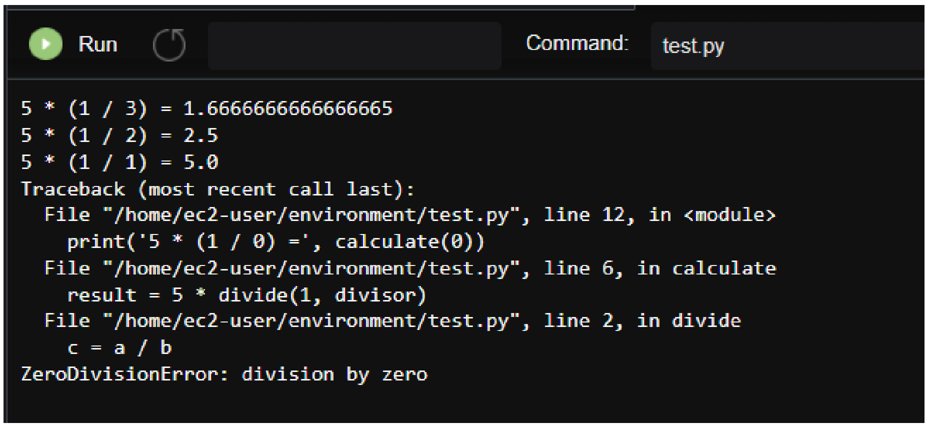
Above an error message like this, one can see what's called the **traceback** or **callstack**. This is a very useful tool for debugging. The traceback shows the position in the code where the error happened. If the error happened inside of a function, it will display both the line inside of the function where the error happened and the line of the code that made the call to the function.

The following code, in a file called test.py, demonstrates how to read a traceback. In this code we first call the calculate function, which in turn calls the divide function, and do so four times for different inputs. The final call on line 12 should run into issues, as we are attempting to divide by zero.



```
1   def divide(a, b):
2       c = a / b
3       return c
4
5   def calculate(divisor):
6       result = 5 * divide(1, divisor)
7       return result
8
9   print('5 * (1 / 3) =', calculate(3))
10  print('5 * (1 / 2) =', calculate(2))
11  print('5 * (1 / 1) =', calculate(1))
12  print('5 * (1 / 0) =', calculate(0))
```

This is the output:



```
5 * (1 / 3) = 1.6666666666666665
5 * (1 / 2) = 2.5
5 * (1 / 1) = 5.0
Traceback (most recent call last):
  File "/home/ec2-user/environment/test.py", line 12, in <module>
    print('5 * (1 / 0) =', calculate(0))
  File "/home/ec2-user/environment/test.py", line 6, in calculate
    result = 5 * divide(1, divisor)
  File "/home/ec2-user/environment/test.py", line 2, in divide
    c = a / b
ZeroDivisionError: division by zero
```

Don’t be intimidated by the callstack, read it from bottom to top. Firstly, the exception is a ZeroDivisionError. That’s clear, we divided by zero somehow. Moving up, we see that the issue happened on line 2 in the *divide*-function, on the line *c = a / b*. Apparently *b* was zero. But how did we get there? Moving up, we see that it was because we made a function call to the *divide*-function on line 6 inside the *calculate*-function. Then how did we get there? Again, moving up, that was from line 12, where we called *calculate(0)*.

In real scenarios, and especially when using external packages, callstacks can end up many layers deep. The whole stack is useful to scan through, but the most important parts are:

- the error type (ZeroDivisionError)
- the error description (division by zero)
- the offending line, deep down in the callstack (line 2 in the *divide*-function)
- the triggering lines up high in the call-stack (line 12 calling *calculate(0)*).

When seeing an error, resist the urge to immediately jump to a guess at a solution and run your code again, lest you introduce further errors or complications. Rather, spend a bit more time looking through the callstack, determine what was wrong, and fix the error correctly in one go. This is a point that deserves your attention as it is where many beginners fail. Programming is exact, guessing will not get you far. If you are overwhelmed by your own error messages the trick is to slow down and do smaller modifications of your code at one time. This might slow you down, but not as much as a faulty debugging process would.

2.11 Imports from Modules and Packages

To structure code for re-use in multiple projects, or within a single large project, Python uses modules and packages. A **module** is a Python file from which we may import functionalities in another Python file. Any Python file can be treated as a module. A **package** is just a group of modules, and may also contain nested subpackages. It is defined through grouping modules together into a directory, and adding an __init__.py file as indicator that it is indeed a package. If a module or a package directory is in the same location as the main file you are running, it can be imported from.

It will not be needed to make packages yourself, but for interacting with Optibook we will use packages that have already been made.

Besides external packages we can install, there are also several built-in modules and packages already in Python which we can import from directly. Among those is the **math module**. The following shows three ways of importing the exponent function *exp* and natural logarithm function *log* from the math package:

```python
# Import the whole math module
import math

print(math.exp(1.0))   # 2.718281828459045
print(math.log(2.7))   # 0.993251773010283
```

```python
# Import only the required functions from the module
from math import exp, log

print(exp(1.0))   # 2.718281828459045
print(log(2.7))   # 0.993251773010283
```

```python
# Aliasing the module name or the function name
import math as m
from math import log as ln

print(m.exp(1.0))   # 2.718281828459045
print(ln(2.7))      # 0.993251773010283
```

The notation to import from a module contained within a package is *<package>.<module>*:

```
from os.path import join
```

For modules in subpackages this sensibly continues as *<package>.<subpackage>.<module>*, and so on. All combinations of the above methods work as expected.
The common practice is to place all import statements at the very top of a file, but they can theoretically be placed anywhere.

A note on a wildcard import functionality you might come across (*'from math import *'*): try to avoid it. It makes it very hard to follow where specific functions come from, and can lead to subtle errors if you import two packages which happen to contain functions with the same name.

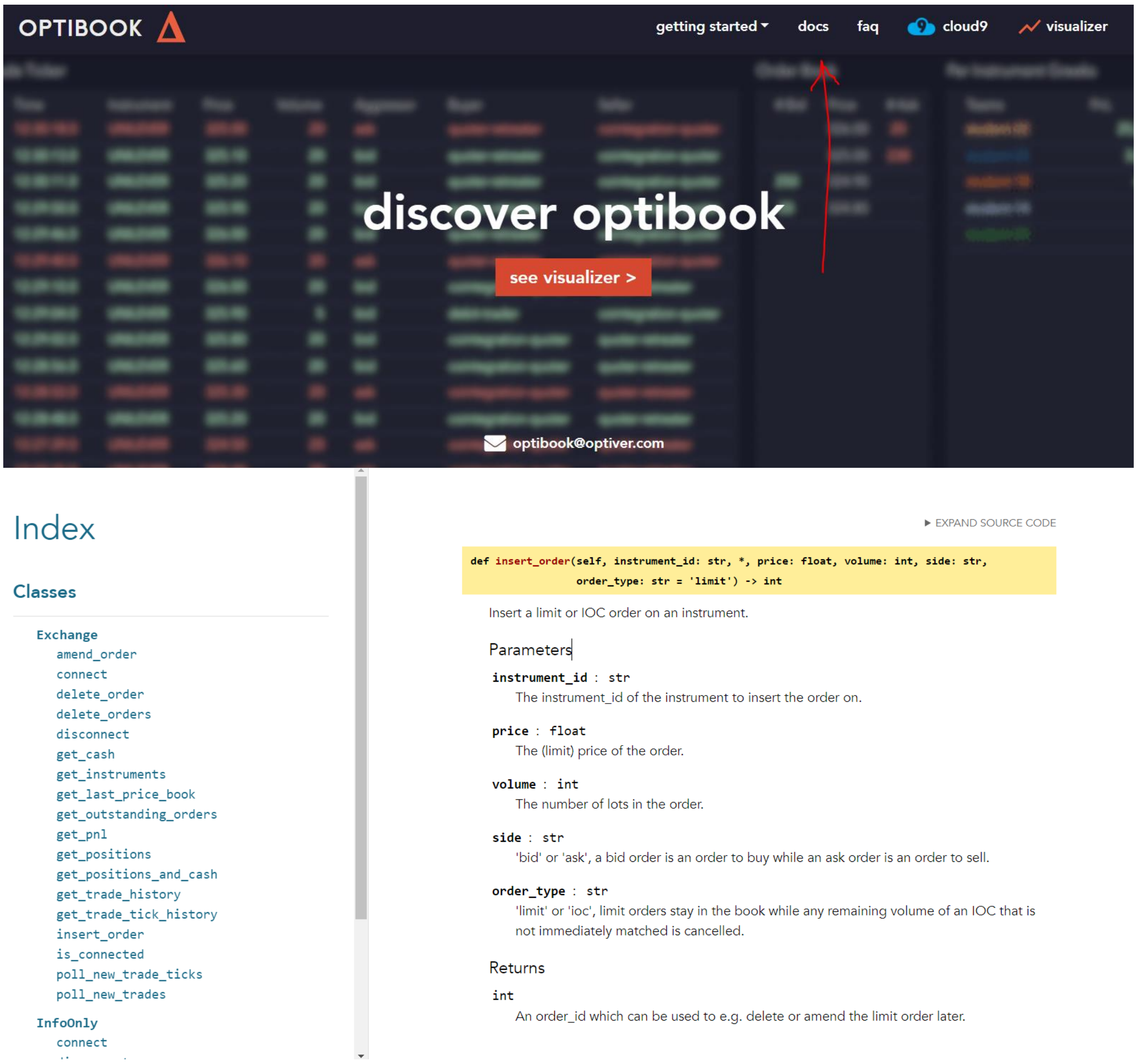
3 Optibook Client

Our **optibook_client** Python package manages the connection to the central exchange, and provides all functionality such as inserting and cancelling orders, receiving price and trade information, etc.

This section gives a brief intro, which is sufficient to write a trading algorithm, but is not exhaustive. Always feel free to play around with any function the Exchange client provides as you see fit, a lot more is possible than the basic algorithms and examples show.

3.1 Finding Documentation

All user-facing functions have appropriate type-hints and docstrings, which are visible through the auto-complete functionality of the IDE or by looking at the source code for each function. There is also full documentation online, found under the docs button on the top-right of the Optibook landing page, see the below screenshots.



Index

Classes

- Exchange
  - amend_order
  - connect
  - delete_order
  - delete_orders
  - disconnect
  - get_cash
  - get_instruments
  - get_last_price_book
  - get_outstanding_orders
  - get_pnl
  - get_positions
  - get_positions_and_cash
  - get_trade_history
  - get_trade_tick_history
  - insert_order
  - is_connected
  - poll_new_trade_ticks
  - poll_new_trades
- InfoOnly
  - connect

```
def insert_order(self, instrument_id: str, *, price: float, volume: int, side: str,
                 order_type: str = 'limit') -> int
```

Insert a limit or IOC order on an instrument.

Parameters

instrument_id : str
The instrument_id of the instrument to insert the order on.

price : float
The (limit) price of the order.

volume : int
The number of lots in the order.

side : str
'bid' or 'ask', a bid order is an order to buy while an ask order is an order to sell.

order_type : str
'limit' or 'ioc', limit orders stay in the book while any remaining volume of an IOC that is not immediately matched is cancelled.

Returns

int
An order_id which can be used to e.g. delete or amend the limit order later.

3.2 Exchange Object

The heart of the package is the Exchange object. It is located in the *synchronous_client* module, and is typically instantiated once at the start of a file containing a trading algorithm. The Cloud9 service is set up such that the exchange connects you under your own username automatically upon calling *exchange.connect()*, as can be seen from the green dot appearing next to your team name in the Exchange Dashboard/Visualizer.

If you log in from two locations or files at the same time, the last place you connect from will take over the connection, and the old connection will be broken.

```
from optibook.synchronous_client import Exchange
```

```
exchange = Exchange()
exchange.connect()
```

All interactions with the exchange happen through methods of this Exchange object.

3.3 Order books

To get the current aggregated **order book** with outstanding bid and ask limit orders, use the following function:

```
book = exchange.get_last_price_book('ASML')
```

This and further examples are applicable to a stock with ticker/instrument_id ‘ASML’, but these are modifiable by providing a different instrument_id.

The returned object is of type *PriceBook*. The **PriceBook type** is a compound type containing two lists as attributes, one of bids and one of asks. The lists of bids and asks are sorted from most competitive to least competitive, so bids are sorted from high to low prices, and asks are sorted from low to high prices. Each of the bids and asks are **PriceVolume types**, which have a price attribute and a volume attribute, which are both just numbers. If there are no bids or asks at all on a side, the corresponding attribute (*.bids* or *.asks*) is None.

```
if not book.bids:
    print('No bids at all for instrument.')
else:
    best_bid = book.bids[0]
    price = best_bid.price
    volume = best_bid.volume
    print(f'Best bid is {volume} lots @ price {price}.')

if not book.asks:
    print('No asks at all for instrument.')
else:
    best_ask = book.asks[0]
    price = best_ask.price
    volume = best_ask.volume
    print(f'Best ask is {volume} lots @ price {price}.')
```

3.4 Tradeticks and Trades

Public tradeticks are the trades by all market participants, including our own. These are public market data, just like the order book, and can provide useful information for a trading algorithm. The trades are recorded starting from the moment you first call *exchange.connect()*; so no additional history is returned.

```
# Get all tradeticks in an instrument (upto max limit)
tradeticks = exchange.get_trade_tick_history('ASML')

# Get new tradeticks since last call
tradeticks = exchange.poll_new_trade_ticks('ASML')
```

The second method, using polling, is useful if you are periodically running a trade loop, to find what new trades have happened, since the last time you checked. The returned type for both methods is a list of **TradeTick objects** sorted on time (newest one last). These are used as follows:

```
if not tradeticks:
    print('No tradeticks happened on instrument.')
else:
    last_tradetick = tradeticks[-1]

    timestamp = last_tradetick.timestamp
    price = last_tradetick.price
    volume = last_tradetick.volume

    print('The last tradetick: ')
    print(f'At {timestamp} {volume} lots traded at price
          {price}.')
```

Private trades are somewhat similar to tradeticks but only refer to our own trades. It is of course useful to not have to sieve through all public tradeticks to find our own, and these are considered private information. The **Trade object** is also different and slightly simpler than the public TradeTick object. When you first call this function, there should be no trades, as you have not inserted orders yet, but similarly to trade ticks, you can get your trade history or poll new trades:

```
# Get all trades in an instrument (upto max limit)
trades = exchange.get_trade_history('ASML')

# Get new trades since last call
trades = exchange.poll_new_trades('ASML')
```

The resulting object is a list of Trade objects:

```
if not trades:
    print('No trades happened on instrument.')
```

```
else:
    last_trade = trades[-1]

    price = last_trade.price
    volume = last_trade.volume
    side = last_trade.side

    print('The last trade: ')
    print(f'We traded {volume} lots at price {price}.')

    if side == 'bid':
        print('We were the buyer.')
    elif side == 'ask':
        print('We were the seller.')
```

3.5 Inserting and Cancelling Orders

To make trades on the exchange, we first need to **insert orders**. If and when these inserted orders match with an opposing order, we will participate in a trade.
On Optibook there are two order types, a **limit order**, which stays on the exchange until it is traded or cancelled, and an **IOC order**, which trades against existing limit orders only, but is instantly removed afterward. That is to say, an IOC order does not stay in the book.

Both limit and ioc orders are inserted through the same method of the *Exchange* object, to which the order type is specified as an argument. There are also other parameters to specify: the price, volume and side. Note that all of the arguments to this function call, except for instrument_id, are keyword-only (see section 2.8.4), so we must specify both the parameter name and value.

Stay aware that there is a discrete ticksize applicable to all products (the minimum price variation). If you insert a non-rounded value for the price, the exchange rounds it to a price exactly on a tick for you.

To cancel a limit order after some time, we must specify its order_id (an integer). The order_id can be obtained in a few ways, but one easy way is to store it when we first insert the order, as it is part of the **InsertOrderResponse object** returned from the *.insert_order()* function.

```
# Insert a limit bid order for 3 lots at the price of 8.0
response = exchange.insert_order('ASML',
                                 price=8.0,
                                 volume=5,
                                 side='bid',
                                 order_type='limit')
```

```
# Cancel the inserted limit order after 2 seconds
import time
time.sleep(2)

exchange.delete_order('ASML', order_id=response.order_id)
```

Another way to track orders it to get all currently still outstanding orders with the following method:

```
orders = exchange.get_outstanding_orders('ASML')
```

The result is a dictionary of **OrderStatus objects** describing the orders, the key to each element is the order_id, and the value is the order status. The OrderStatus objects have attributes *.order_id*, *.instrument_id*, *.price*, *.volume* and *.side*. Combining these, we can get to the following useful code snippet, which cancels all outstanding orders:

```
orders = exchange.get_outstanding_orders('ASML')
for order_id in orders.keys():
    exchange.delete_order('ASML', order_id=order_id)
```

Finally, it's possible to delete all outstanding limit orders for an instrument using the *.delete_orders()* method.

```
exchange.delete_orders('ASML')
```

3.6 Positions

The following methods allow for loading your own current **position**. The resulting object is a dictionary, its keys are instrument_ids and the values are integers giving the amount of lots you hold in that instrument. You can count on all exchange-defined instruments to always be present in the dictionary, even if you do not have a position in them.

```
# Get current positions in all instruments
pos = exchange.get_positions()

for iid in pos.keys():
    print(iid, pos[iid])
```

3.7 PnL

Finally, there is a method to obtain the current **PnL**. The calculation is based on the last traded price of each of the instruments, be mindful that if an instrument trades very little, this could deviate from current market bids and asks. The function returns a float:

```
print(exchange.get_pnl())
```

The valuation of each position for your PnL calculation is by default based on the last public traded price for that instrument. It is possible to override any particular valuation by providing a dictionary of overrides as argument:

```
print(exchange.get_pnl({'ASML': 10.0}))
```

This would evaluate the ASML stock as being worth exactly 10.0, while valuing any other positions based on the last traded price as before.

4 Optibook Exporter

Our **exporter** Python package allows you to export data in a fixed format with synced timestamps between files. This allows you to log or store information while running your algorithm.

4.1 Setting up the exporter

In order to use the exporter, you need to import it first:

```
from optibook.exporter import Exporter

exporter = Exporter()
```

4.2 Exporting information

The process for exporting instrument data is initiated by creating a structured log request. The log request is structured as a dictionary where each key represents a file name, and the corresponding value is a list of lists. The outer list represents a row in the export, the inner row represents a column. This structure allows for the definition of variable data, organized in one or more files.

In the example below, we want to log 3 instruments and their prices to a file called *instruments.csv:*

```
logrequest = {
    "instruments.csv": [
        ["Instrument A", "Price A"],
        ["Instrument B", "Price B"],
        ["Instrument C", "Price C"],
    ]
}
```

We then log the request as follows:

```
exporter.export(logrequest)
```

The result of this will be a file called *instruments.csv* in the *exports* folder. The created file contains 3 rows, all having 3 columns: a timestamp, the instrument name and the instrument price. The timestamp is added by the exporter and will be the same on every row.

4.3 Resetting files

Resetting files can be done by deleting the log files in *exports* folder or programmatically using:

```
exporter.export(logrequest)
```