IT security

(VIHIAC07)

Extra material

Python Handbook

Author:
Gábor Fuchs

February 13, 2025

Contents

1 Introduction

This is a guide for the Python 3 programming language, for those, already experienced with other languages.

Python is high level a language, which is meant to be easy to read, write, and execute.

```python
print('Hello, scripting world!')
```

Its laconic syntax, dynamic typing and interpreted nature makes Python a great choice for quick automation of simple tasks, and an also available, but not so great choice for larger projects.

Most mistakes that are caught compile time in workflows with compiled languages, will unfortunately cause run time errors and bugs with Python. In return, we have an extremely powerful tool to look inside what is going on run time: the interactive interpreter shell. Even for learning the language: just start an interpreter shell and start experimenting with what you learn.

2 Working with Python

2.1 Installation, package management, virtual environments

Installing the Python 3 bundle is quite straightforward on all common desktop operating systems. Whichever of these systems you use, you should easily get to have `python3` in the path of your operating system command line shell. This means that you can execute the `python3` command opening a Python interpreter shell whatever your current working directory is. In most usual setups you will also have `python` as an alias to the same program. You can make sure of this by running both `python3 --version` and `python --version`, and checking if both commands work and their outputs are the same.

You can exit a Python shell by typing `exit` or `quit`, or sending end of file to its console input.

Python comes with its own package manager called `pip`. `pip` can install packages from the Python Package Index, a huge repository of public Python packages. These packages include libraries that we can use in our programs, and some programs that are executable on their own. Installed executable packages can be run by `python3 -m <name of the package> <arguments to the executed package>`.

One the packages that are installed by default in recent distributions of Python is the `venv` package. `venv` can create so called *virtual environments*. If you create such an environment and *activate* it in an instance of your OS shell, it will prepend a directory specific to the given environment to your OS shell path with copies of Python related executables. Now if you run `python` or `pip` within this shell, it will use a separate world of installed packages. This is useful because you can easily recreate a Python environment with a specific versions of a specific set of packages installed.

It is also possible not to use virtual environments and install packages system wide (or for a given user). In many systems, the system's package manager (like `apt` on Ubuntu Linux) takes over this responsibility. As these OS package managers might provide a different set of Python packages, or different versions of some packages, I would recommend sticking to virtual environments and `pip` whenever possible.

Let's create a virtual environment and install a package within.

```
python3 -m venv my_venv
```

This created a directory within your current working directory called `my_venv`, which has all the files of the new environment within. Feel free to take a look. Note that some of the generated scripts within may contain the absolute path to this directory hard coded, so do **not** move directories containing virtual environments and expect them to work. Of course you can update all the occurrences of the absolute path within the scrips if you really must move an existing environment, but its usually easier to create a new one and install the same packages as shown towards the end of this section.

To now activate the environment we have to run a script from somewhere in the directory called something like `activate`. The exact script and how to run it actually depends on the operating system and shell you are using. The two most likely cases:

Unix-like systems + bash or zsh:

```
source my_venv/bin/activate
```

Windows + PowerShell:

```
my_venv\Scripts\Activate.ps1
```

In case PowerShell does not let you run the script, you may need to change your active *execution policy* (see the documentation of `venv`).

For other setups, see the documentation.

Note that this only activates the virtual environment for the shell instance you run this command within. Usually a virtual environment being active is indicated by its name being included in your command prompt that appears before the commands you type into the shell. To deactivate the virtual environment simply close the shell and open a new one, or run the command `deactivate`.

As an example to how to install a package within your activated environment, I show how to install the IPython package. IPython provides an enhanced interactive shell experience with tab completion, syntax highlighting and other convenience features.

```
pip install ipython
```

Now you can run the IPython interpreter shell by simply executing the `ipython` command while the environment having it installed is active. Another way is executing it just like we executed `venv` before: `python -m IPython`.

To list the dependencies of a project so that they can be easily installed later in any environment, you can create a file most frequently named `requirements.txt` containing all the names of the required packages. Such a file may or may not include versions to the listed packages. You can install all the listed packages within such a text file by executing `pip install -r requirements.txt` while having the environment activated where you need them.

To generate the content for a `requirements.txt` of all the packages installed within an environment together with their exact versions, you can use `pip freeze` command. Note that `pip freeze` will not only list explicitly installed packages, but their dependencies too. In most shells you can directly write the output to a file called `requirements.txt` by `pip freeze > requirements.txt`. By creating a `requirements.txt` this way and installing the listed packages in a newly created environment by `pip install -r requirements.txt`, you can easily migrate your virtual environment to another location in your file system, or even other machines possibly running different systems.

2.2 Running Python code

You have quite many ways to run code written in Python.

The simplest option is to open an interactive interpreter shell and enter lines of directly, which will run instantly as you hit enter. As shown earlier, you can start the default interpreter by running the `python` program, or install and use the `ipython` program for an enhanced experience.

Your next option is to explicitly ask `python` to execute a Python source file. For example we can create a file called `my_python_script.py` with the following content:

```
print('Hello, scripting world!')
```

and execute it by

```
python my_python_script.py
```

We usually refer to short single file programs written in interpreted languages like Python as scripts.

A fun way to combine the above two options is to use the `-i` argument when passing your program to `python` (also supported by `ipython`):

```
python -i my_python_script.py
```

This runs the program first, and then opens an interpreter shell still having all the global variables after running the program available just like if you pasted the lines of the program into the interpreter one-by-one and still have it open. This can be quite useful to write scripts incrementally.

I would also mention the `%history` command in the IPython shell here, which outputs a copyable list of the previous lines of code you passed to the interpreter. I often find myself running a lot of lines of code in an interpreter and deciding to make a script from my calculations so I can reproduce it later, or edit it easier as a file. In those cases this command is really useful.

On Unix-like systems you can easily make a Python source file actually executable “in itself” by adding a so called *shebang* line starting with `#!` to its very beginning:

```
#!/usr/bin/env python3

print('Hello, scripting world!')
```

This is a reference to the interpreter program the OS needs to feed the rest of the file into to semantically execute it, so this will of course still require Python to be installed on the system (and the virtual environment activated if applicable). As `#` is used for comments in Python (just like in many other scripting languages), from Python’s point of view, this line does not do anything, so you can still run it as before by passing the file to the `python` program as a command line argument as before. The shebang reference needs to be an absolute path, like `#!/usr/bin/python3`, but hard coding it as such would break the use of virtual environments. The use of the `env` program (`/usr/bin/env` as above) is used to solve this issue. Having added the shebang line, we just need to make sure that we have permission to execute the file (which is not required, when manually passing it to the interpreter as in `python my_python_script.py`).

```
chmod +x my_python_script.py
```

Now we can simply execute it:

```
./my_python_script.py
```

3 The Python language

3.1 Code lines and code blocks

The most iconic features of Python's syntax is marking code lines by line breaks alone (i.e. no `;` at the end of each line, although you can use it to put multiple simple statements not involving code blocks on the same physical line of code), and marking code blocks by indentation alone (no `{ ... }` surrounding code blocks). This affects the syntax of even the most basic elements of the language like if-else, loops and function definitions. Compare the following mostly corresponding pieces of C++ and Python codes.

```
template<typename T>
T my_max(T arg1, T arg2) {
    // body of my function
    if (arg1 > arg2) {
        return arg1;
    } else {
        return arg2;
    }
}
```

```
def my_max(arg1, arg2):
    # body of my function
    if arg1 > arg2:
        return arg1
    else:
        return arg2
```

In this above example we can examine three code blocks: A larger block containing the entire body of the function `my_max`, and two smaller blocks within containing the bodies of the `if` and `else` branches. All blocks have some kind of header line before their body, ending in a `:` character. The body of the block is simply marked by a deeper level of indentation, i.e. more spaces/tabs before the start of the line of code, and ended by a line with less deep level of indentation (no longer belonging to the body of the block) or the end of the file.

A few notes on these code blocks:

- You can use any combination of spaces and tabs for the indentation. I recommend the usual style of 4 spaces per level of indentation, but you can technically mix them in all kinds of weird ways to achieve horrible code quality. The only thing that really matters is that the interpreter can decide if two subsequent lines of code (lines containing whitespace characters and comments only do not matter) …
    - have an equal level of indentation,
    - the second one is on a deeper level of indentation (the previous (first) line must have ended with `:`, start of a block), or
    - the second one has a smaller level of indentation (must be equal to one of the previous levels, ending all the in-between level active blocks since the level we return to was last used).
- If the code block contains only a single line, not involving any further code blocks, you have the option to write it in the same line as the block header (e.g. `if arg1 > arg2: return arg1`, can actually be multiple such statements divided by `;`-s).
- All blocks must contain at least one line. If you need a semantically empty block, you must use the special keyword `pass` as a placeholder for the content of the block, which in turn can only be present if it is the only thing in the given block, i.e. if the block really is empty. (Note that `pass` is not a "do nothing" or "noop" instruction, but a core keyword of the language used with all kinds of code blocks, e.g. it is also used in class definitions without members.)

```python
def do_nothing():
    pass
def do_nothing2(): pass # `pass` can also be written after the block header
```

3.2 Top level scripting

In contrast to some other languages, in Python you do not have to define a `main` function, to write a program executable in itself. In fact, even if you decide to follow the relatively common practice of creating a `main` function, which can be useful to enable `main` having some local variables (see the next section about variable scope), you still have to call it from the top level to make it actually run.

```python
def main():
    print('Hello')

main() # actually call the `main` function
```

All the code you write on the top level runs immediately when the file is read by the interpreter, both if it is directly run as program and if it is imported by other code. This can be useful even when writing a library, for example to initialize global variables.

A common practice is to write Python source files that can both be executed as a program and used as a library from another with different behavior. You can achieve this by checking the name of the current module (`__name__`) run time, which will be `'__main__'` if the file is directly executed.

```python
def main():
    ...

if __name__ == '__main__':
    main() # call our main function
```

3.3 Functions, variables and scope

You can create new variables in a given scope by simply assigning a value to them with the `=` operator.

```python
x = 3
print(x) # prints `3`
```

Side note: There is actually another operator for assigning values, the so called Walrus operator `:=`, which lets you assign values as part of an expression by returning the assigned value:

```python
a = (b := 3) * 2
print(a, b) # prints `6 3`
```

A huge difference to most other languages are the scope and lifetime of these variables. In other languages usually every code block, even the body of an if statement serves as a separate scope for variables, meaning that variables declared within such a block cannot be accessed from outside the given block. In Python, only bodies of functions create separate scopes.

```python
def my_func(condition):
    if condition:
        x = 3
    else:
        x = 5
    return x
```

In most other languages, we would need to explicitly declare `x` before the `if`, outside its body to make its lifetime reach the `return` statement outside of the body of the `if` end `else`. In Python, however, due to its function level variable scopes, setting it in any of the branches results in having the variable implicitly declared as a local variable for the entire function, from its very beginning. This declaration only holds the information "`x` is a local variable". Due to the dynamic typing used by the language, the type of `x` is stored together with its value, and can even change whenever a new value is assigned of a new value. In fact, while the first assignment has not happened yet, the variable is undefined, and does not have either a value or a type.

```python
def my_func2(y):
    if y < 0:
        x = 3
    elif y > 0:
        x = 'this is a string instead of an int'
    else:
        pass # do not even set any value
    print(x)
```

The call `my_func2(-1)` will print the integer `3`, `my_func2(1)` will print the string `'this is a string instead of an int'`, and `my_func2(0)` will result in a run time exception `UnboundLocalError: cannot access local variable 'x' where it is not associated with a value`.

3.3.1 Variables from outer scopes, shadowing

If a variable with a given name is set anywhere in the function (even if that line is not executed do to a run time condition like in case of `my_func2(0)`), any mention of it will be treated as reference to a local variable, causing the above exception if it is read before being actually written. Any variables of outer scopes with the same name are shadowed, so the same exception will be raised if the local variable is not yet defined, even if a variable with the same name exists in an outer scope (e.g. a global variable, see `my_func4` below).

```
x = 2
def my_func3():
    print(x) # reads the global variable
def my_func4():
    # print(x) would error here as there is a local variable
    #          called `x`, but it has not yet been defined
    x = 4
    print(x) # reads the local variable

my_func3() # prints `2`
my_func4() # prints `4`
print(x) # prints `2`
```

If a variable is only read but never set within a function (eg. `my_func3`), the resulting behavior is different: the interpreter looks for the variable in the outer scopes until it finds it. If such variable does not exist in any of the outer scopes, this results in a different exception raised: `NameError: name 'x' is not defined`.

This leaves one question: how to set a variable of an outer scope? To override the default behavior of implicitly declaring a local variable and shadowing the variable of the outer scope, when setting the variable anywhere in a function, we have to explicitly declare the given variable as an outer variable. Two different keywords can be used for this purpose: `global` and `nonlocal`. `global` will always look for the variable in the top level scope containing global variables, while `nonlocal` will replicate the behavior used when the variable is only read, finding the first variable with the given name in the inner most outer scope where such a variable is declared (i.e. having it set anywhere without declaring it `global` or `nonlocal` in the same scope).

Let's examine all these behaviors on an example with nested functions.

```
x = 3 # global `x`
def test_function():
    x = 4 # x local to test_function
```

```python
    def inner_function0():
        x = 6 # setting a local variable `x` of this nested function,
              # has no effect
    def inner_function1():
        global x
        x = 7 # setting the global variable `x` from `3` to `7`
    def inner_function2():
        nonlocal x
        x = 8 # setting the local variable `x` of test_function
              # from `4` to `8`

    inner_function0()
    inner_function1()
    inner_function2()

    print('local x of test_function:', x) # will print `8`

test_function()
print('global x:', x) # will print `7`
```

3.3.2 Undefined variables, the `del` keyword and the `None` value

Assignment of a variable associates the variable with a type and a value. If for some reason we need to return the variable to its original undefined state, we can "delete" it, with the `del` keyword.

```python
# print(x) would error, x is not defined
x = 5
print(x) # prints 5
del x
# print(x) would error, x is not defined
```

Note that there is an other use of the `del` keyword for deleting elements from certain compound types (see later).

As discussed earlier, a variable being undefined has nothing to do with it being declared as a local variable in a function, so deleting a local variable will not make the interpreter look for variables in outer scopes.

Note that unlike in the case of some other languages (e.g. `nil` in Lua), being undefined is completely different from a variable holding the `None` value, which is used as a placeholder for a semantically absent value, for example returned by functions without an explicit `return` statement.

3.4 Types

Let's get to know the core types of the Python language, and how types are handled in general in the following subchapters.

3.4.1 Primitive types: `int`, `float`, `bool`; and operations with them

Some languages, like Java make a clear distinction between primitive (lower case named `int`, `float`, etc.) and reference types (typically upper case named, instances of classes). Primitive types usually have a different behavior when passed to functions as arguments: the primitive value itself is copied, while in case of reference types, only a reference (pointer, memory address) to the existing "object" is copied, not the contained data itself.

Python does not make such distinction. All types are reference types in this sense (and are defined by classes), however some types are *immutable*, which in practice results in a similar behavior as what we might be accustomed to with primitive types in other languages. If you pass an immutable object to a function, you can be sure that the object itself cannot be changed by that function:

```python
def my_func(x):
    print(x) # prints 1
    x = 2
    print(x) # prints 2

my_var = 1
my_func(my_var)
print(my_var) # prints 1
```

So everything seems to work just like with actual primitive values in other languages, but what happens under the hood is:

1. The global `my_var` variable is a reference to an integer object representing the value `1`.
2. When `my_func` is called with `my_var`: the reference is copied to the local variable `x`, which will point to the same object in the memory.
3. The local variable `x` is set to `2`: a new integer object is instantiated representing the value `2`, and now `x` points to this new object, while the global `my_var` still points to the original object representing `1`. The value of the existing object cannot be changed as the `int` type is immutable, meaning it does not provide an interface to do that.

   Side note: For some integer values (especially small ones like `1` and `2` above) you might find that certain Python environments do not actually instantiate new objects every time an expression evaluates to the given value, but rather reuse some cached objects representing those values.

In turn for integers being objects, which obviously causes an overhead both in memory footprint and speed compared to an actual "primitive" type, we get something nice: they can store arbitrary large numbers! This means we do not need to worry about errors caused by overflows. Try typing `2**2000` (2^{2000}) into a Python shell. You could not fit that number in any conventional fix sized integer.

Let's take a look at common operations with basic types.

Booleans (`bool`):

```python
t = True
f = False
assert t
assert not f
assert t and not f
assert not (t and f)
assert t or f
assert t or t
assert not (t == f)
assert t != f
assert t ^ f
```

An `assert` statement takes an expression (syntactically just like `return`), and raises an exception (see about exceptions later) if the evaluated value is *falsy* (and not *truthy*), meaning it becomes `False`, when converted to `bool`; otherwise it does nothing. I will use it to show expected values of expressions.

Note the syntax of the above literals and operators. `True` and `False` are the literals for the two different possible values of booleans. For logical operators Python uses the English words `not`, `and` and `or` (instead of `!`, `&&`, `||`).

Operators `==` and `!=` are Python's *equals* and *not equals* operations. Types can define what their instances are equal and not equal to. If you want to check if two variables reference the exact same object, you can use the `is` and `is not` operators. This is useful, when checking if a value is `None`, which is a singleton object or when checking the exact type of a value as Python represents each type by an object (of type `type`):

```python
a = None
assert a is None
b = False
assert type(b) is bool
```

Another way to check `a is b` would be `id(a) == id(b)`. IDs are integers unique to each object in the memory, much like a memory address (can actually be the memory address in certain implementations).

Side note: `bool` is actually a subclass of `int`, which allows some weird operations, like counting `True` values in a collection of booleans by summing them, as `True` corresponds to the integer value `1`, and `False` corresponds to `0`.

```python
assert type(True) is bool
assert isinstance(True, bool)
assert isinstance(True, int)
assert True + True + False == 2
assert True > False
```

`isinstance` allows checking for superclasses as well, not only exact types

You can use a couple of different literal formats for integer values:

```python
assert 0b100 == 4
assert 0x100 == 256
assert 0o100 == 64   # 0100 in some other languages,
                     # writing 0100 is not allowed at all in Python
                     # to avoid confusion
assert -0b10 == -2
```

You can use _ between digits to segment your literals for better readability:

```
assert 0b1000_0100_0101_1111 == 0x845f
assert 999_999 + 1 == 1_000_000
```

The most basic operations with integers (`int`), work just like in other languages:

```
assert 1 + 1 == 2
assert 1 - 2 == -1
assert 1 * 1 == 1
```

Integer division is done by `//` and modulus by `%`:

```
assert 5 // 2 == 2
assert 5  % 2 == 1
```

An interesting difference in how these operations work compared to other languages, is that they represent Euclidian division and modulus, which give a bit different results when working with negative numbers:

```
assert -5 // 2 == -3   # would be -2 in some other languages
assert -5  % 2 == 1    # would be -1 in some other languages
```

The operator `/` always does floating point division:

```
assert 5 / 2 == 2.5
assert 4 / 2 == 2.0
assert type(2) is int
assert type(2.0) is float
assert type(4 / 2) is int
assert type(4 / 2) is float
```

The operator `**` calculates powers:

```
assert 2 ** 3 == 8
assert type(2 ** 3) is int
assert 9 ** 0.5 == 3.0  # a simple way to calculate square root
                        # without importing anything
assert type(9 ** 0.5) is float
assert 2 ** -1 == 0.5
```

You can also use the builtin `pow` function with two arguments, to achieve the same:

```
assert pow(2, 3) == 8
assert pow(9, 0.5) == 3.0
assert pow(2, -1) == 0.5
```

I mention this only because you can also use `pow` with 3 integer arguments, which gives you power over a finite field of integers:

```
assert pow(2, -1, 7) == 4  # multiplicative inverse of 2 over GF(7) is 4
                           # (i.e. (2 * 4) % 7 == 1)
```

Numeric values can be compared with `<`, `>`, `<=`, `>=` operators additionally to `==` and `!=`. As interesting additional feature, all these operators can be used together in chains like `a != b == 3 < 4`, which is equivalent to `a != b and b == 3 and 3 < 4`. This is a useful shorthand for example when checking if a value fits within an interval: we can write `3 <= x < 5` instead of `x >= 3 and x < 5`.

We also have the usual bitwise operators on integers:

```
assert 6 ^ 3 == 5       # bitwise xor
assert 6 & 3 == 2       # bitwise and
assert 6 | 3 == 7       # bitwise or
assert 3 << 2 == 12     # bit shift leftwards
assert 3 >> 1 == 1      # bit shift rightwards
assert ~3 == -4         # bitwise not
```

Interestingly the bitwise operators except for `~`, but including `^` do return `bool` values and thus operate as logical operators, when both operands are `bool`-s. This is possible because types can overload the behavior of these operators (as opposed to the logical operators).

Most operations have an assignment version of them, so `x = x + 1` can be written as `x += 1`, or `x = x << 1` as `x <<= 1`. Note that in case of working with immutable types (like all the above "primitive" types), these operations still replace the object just like the explicit `x = x + 1` syntax shows, not change the existing one. Mutable types might decide to change the existing object when using `+=` and similar operators with them.

3.4.2 Core compound types: `tuple`, `list`, `set`, `dict`

Python has a couple of builtin compound / collection types.

```
t = (1, 2.0, False)
assert type(t) is tuple
l = [1, 2.0, False]
assert type(l) is list
s = {1, 2.0, False}
assert type(s) is set
d = {1: True, 2.0: 3, False: 5}
assert type(d) is dict
```

Due to Python's dynamic typing all of these collections can hold items of different types at the same time. Just like any defined variable actually contains a reference to object, collections contain only references to their items. Note that all the above enumerations of elements support trailing commas (a comma after the last element as well).

```
l2 = [
    1,
    2,
    3,
]
```

All of the above collections support calling the builtin `len` with them to get the number of items contained within them:

```
assert len(t) == 3
assert len(l) == 3
assert len(s) == 3
assert len(d) == 3
```

Tuples are immutable, so you cannot add or remove items from them after being created. Of course, if the contained objects are **not** immutable, they can change, but cannot be replaced for other objects.

Lists on the other hand are mutable, so items can be added, removed and replaced any time.

```python
t = (1, 2.0, False)
assert t[0] == 1
assert t[2] == False
l = [1, 2.0, False]
assert l[0] == 1
assert l[2] == False
l.append(t)
l[0] = -1
assert l == [-1, 2.0, False, (1, 2.0, False)]
```

In implementation, tuples and lists are similar to vectors or array lists in other languages, so it is fast to access items by their index, but it is slow to insert or remove items from the beginning or middle of the collection. You can check if a value is present within a `tuple` or `list`, with the `in` and `not in` operators, but this is also slow as it requires to check all the contained items:

```python
l = [1, 2, 3]
assert 4 not in l
assert 2 in l
```

You can create an empty tuple by `()`, but a one-item tuple need a trailing comma after the item `(1,)` otherwise it would be interpreted as simple parentheses around the expression.

An interesting feature of indexing in Python is the usage of negative indices, which allows easy referencing items based on their relative indices to the end of the collection. `[-1]`

```python
l = [10, 11, 12]
assert l[-1] == 12
assert l[-3] == 10
assert l[-1] is l[len(l) - 1]
```

Note that indexing does not loop further in any direction, so indices must be between `-len(l)` and `len(l) - 1`.

Another feature is indexing parts of a tuple or list:

```python
t = (10, 11, 12, 13)
assert t[1:3] == (11, 12)
assert t[1:-1] == (11, 12)
assert t[-3:-1] == (11, 12)
assert t[1:4] == (11, 12, 13)
assert t[1:] == (11, 12, 13)
assert t[:3] == (10, 11, 12)
assert t[:] == t
l = [10, 11, 12, 13]
assert l[:-1] == [10, 11, 12]
assert l[1:1] == []    # zero items
assert l[2:1] == []    # would be negative number of items, still zero
                       # (no exception raised)
```

You can even replace a part of the list even with a different number of items:

```
l = [10, 11, 12, 13]
assert l[1:2] == [11]
l[1:2] = [10.5, 11.5]
assert l == [10, 10.5, 11.5, 12, 13]
```

An even more interesting indexing, is providing a third number when indexing indicating steps:

```
l = [10, 11, 12, 13, 14]
assert l[0:5:2] == l[::2] == [10, 12, 14]
assert l[-1:-6:-1] == l[-1::-1] == l[4::-1] == l[::-1] == [14, 13, 12, 11, 10]
```

Even such so called *extended slices* can be replaced, but only with a matching number of items:

```
l = [10, 11, 12, 13, 14]
l[::-2] = [1, 2, 3]
assert l == [3, 11, 2, 13, 1]
```

You can concatenate both lists and tuples using the `+` operator.

```
assert (1, 2) + (3, 4) == (1, 2, 3, 4)
assert [1, 2] + [3, 4] == [1, 2, 3, 4]
```

You can also use `+=`, but note that in case of tuples, which are immutable, it replaces the object with a new one, while in case of lists, it actually changes the original list:

```
t1 = (1, 2)
t2 = t1
t2 += (3, 4)
assert t1 is not t2
assert t1 == (1, 2) and t2 == (1, 2, 3, 4)
l1 = [1, 2]
l2 = l1
l2 += [3, 4]
assert l1 is l2
assert l1 == l2 == [1, 2, 3, 4]
```

You can make repeated versions of a tuple or list by multiplying them by an integer using the `*` operator. Except similar behavior as above with `*=`.

The third (and possibly least frequently used of the four) builtin collection type is `set`. Sets can only contain *hashable* items. This means that they can be asked to map themselves to an integer value, based on which `set` can assign them hash buckets. Within a hash bucket two elements are considered the same, if they are equal (`==`). What is important to us, is that all the primitive types discussed in the previous section, `str` and `bytes` discussed in the next section are all hashable, as well as tuples of hashable items only. Sets can contain unique values:

```
s = {1, 5, 5}
assert s == {1, 5}
assert 6 not in s
s.add(6)
```

```python
assert 6 in s
s.add(6)
assert s == {1, 6}
```

Adding elements to a `set` or removing elements from it are both fast, as well as checking if an element is present within the collection.

As `{}` is reserved for the more commonly used empty dictionary, you can create an empty set by calling the constructor `set()` without arguments.

Dictionaries (`dict`) are storages of key-value pairs. Dictionaries work quite similarly to sets, with the difference that there is an additional value bound to each hashed value (key).

```python
d = {1: 10, 2: 20, 2: 30}
assert d == {1: 10, 2: 30}
assert 10 not in d # looks for 10 as a key
assert d.get(10) == None # reading d[10] would raise an exception
d[6] = 16
assert d.get(6) == 16
assert d[6] == 16
assert 6 in d
d[6] = 17
assert d == {1: 10, 2: 30, 6: 17}
```

As another use of the `del` keyword, indexable mutable collections might allow deletion of their items based on their indices:

```python
l = [1, 2, 3, 4, 5]
del l[2]
assert l == [1, 2, 4, 5]
del l[1:-1]
assert l == [1, 5]

d = {1: 2, 3: 4}
del d[1]
assert d == {3: 4}
```

3.4.3 Strings and bytes

Strings (`str`) are pieces of text constructed of Unicode characters, while `bytes` holds a "string" of arbitrary byte values. Both are immutable.

For string literals you can use both `'single quotes'` and `"double quotes"`, without any difference. The only practical difference is that you can use double quotes in strings delimited with single quotes without escaping and vice versa. For their slightly cleaner look single quotes are more often used.

Of course, you can escape all kinds of characters in string literals (`'\'\\'`) and use escape sequences (`'\n\t'`), just as in string literals of other languages. Python also has a *raw literal* format prefixed with `r`, without these escapes to enable easily copying strings unintentionally containing such sequences to your code:

```python
assert '\\\'' == r'\''
```

Note that this raw format has some limitations.

Python supports multiple format string formats. The syntax required by the `format` method uses pairs of `{}` as placeholders, inside of which you can place options regarding how the substituted value should be formatted. A shorthand for using this format is prefixing the literal with `f`, and including the expressions inside the `{}` within the format string.

```
assert '1 + {} = {}'.format(1, 1 + 1) == '1 + 1 = 2'
assert '1 + {} = {:02}'.format(1, 1 + 1) == '1 + 1 = 02'
assert '1 + {} = {:.3f}'.format(1, 1 + 1) == '1 + 1 = 2.000'
assert f'1 + {1} = {1 + 1}' == '1 + 1 = 2'
assert f'1 + {1} = {1 + 1:02}' == '1 + 1 = 02'
assert f'1 + {1} = {1 + 1:.3f}' == '1 + 1 = 2.000'
```

Python also supports the format string format used by the `printf` C function, with the `%` operator between the format string and a tuple containing the arguments:

```
assert '(1 + %d) / 100 = %4.1f%%' % (1, 1 + 1) == '(1 + 1) / 100 =  2.0%'
```

For `bytes` literals use the prefix `b` as in `b'abc'`, which represents the 3 bytes with values `0x61`, `0x62`, `0x63` following each other. Byte values without corresponding ASCII characters are usually marked as `\xhh` substituting `hh` with the byte value in hex format.

```
assert b'\x61\x62\x63\xff' == b'abc\xff'
```

All of the literal types mentioned so far (`''`, `r''`, `f''`, `b''`) have multi-line versions delimited by triple single or triple double quotes:

```
assert '''line1
line2''' == 'line1\nline2'
assert f"""line{1:02}
line{2:02}""" == 'line01\nline02'
```

As Python has no syntax for multi-line comments, some people use multi-line strings as hack to "comment out" larger portions of code. However, this way it will still be interpreted as an unnecessary multi-line string, so I would rather recommend using your IDE's builtin shortcut to comment or uncomment selected regions of code (maybe `ctrl`+`/`), which inserts or removes `#` from the beginning of every line.

You can convert between `str` and `bytes` using the `encode` and `decode` methods. These methods optionally take the name of an encoding as an argument, which defaults to `'utf-8'` if not provided.

```
assert 'naïve'.encode() == b'na\xc3\xafve'
assert 'naïve' == b'na\xc3\xafve'.decode()
assert len('naïve') == 5
assert len(b'na\xc3\xafve') == 6
```

You can get the length of both a `str` or `bytes` object by calling the `len` function with them as an argument. Some Unicode characters are encoded on multiple bytes and not all sequences of bytes are valid utf-8. Trying to decode a byte sequence that is not valid in the given encoding will result in an exception raised.

By indexing `str` we get `str` objects, even if we refer to a single character:

```
ii = 'naïve'[2]
assert ii == 'ï' == 'naïve'[2:3]
```

```python
assert type(ii) is str
assert len(ii) == 1
aiiv = 'naïve'[1:-1]
assert aiiv == 'aïv'
assert type(aiiv) is str
```

By indexing `bytes` you get integer values between 0 and 255:

```python
assert b'abcd'[1] == ord('b') == 0x62
assert type(b'abcd'[1]) is int
assert b'abcd'[1:2] == b'b'
assert type(b'abcd'[1:2]) is bytes
```

The `ord` function maps single character strings to the Unicode code of the given character.

`bytes` has a mutable version called `bytearray`, which can be similarly edited to a `list`, with the additional constraint of its items being strictly integers between `0` and `255`. You can convert between them via their constructors as shown in the next section.

To convert integers to their binary representation, you can use the `from_bytes` static method and `to_bytes` method of the `int` type:

```python
assert (1).to_bytes(3) == b'\x00\x00\x01'
assert (1).to_bytes(3, 'little') == b'\x01\x00\x00'     # little endian
assert (-2).to_bytes(3, signed=True) == b'\xff\xff\xfe' # two's complement
    signed
assert int.from_bytes(b'\x01\x00\x00') == 0x01_00_00 == 65536
assert int.from_bytes(b'\x01\x00\x00', 'little') == 1
assert int.from_bytes(b'\xff\xff\xfe') == 0xff_ff_fe == 16777214 == 256 ** 3 -
     2
assert int.from_bytes(b'\xff\xff\xfe', signed=True) == -2
```

You can concatenate `str` or `bytes` objects with the `+` operator, and even make them repeat by multiplying them with integers:

```python
assert 'ab' + 'cd' == 'abcd'
assert 'ab' * 3 == 'ababab'
assert b'ab' + b'cd' == b'abcd'
assert b'ab' * 3 == b'ababab'
```

Python has lots of methods on `str` and `bytes` for string manipulation. I would recommend knowing at least `strip`, `split`, and `join`:

```python
txt = ' aa bb cc '
assert txt.strip() == 'aa bb cc'
words = txt.strip().split()
assert words == ['aa', 'bb', 'cc']
assert ', '.join(words) == 'aa, bb, cc'
```

`strip` and `split` can take both take a `str` as parameter controlling what characters they remove (and cut along). Note that in case of `strip` this parameter (`chars`) is handled as a collection of characters to be removed, so their order does not matter, while `split` handles it (`sep`) as a potentially multi-character separator:

```python
assert 'aaabbbcccaaabbb'.strip('ba') == 'ccc'
assert 'aaabbbaaabbb'.split('ab') == ['aa', 'bbaa', 'bb']
```

3.4.4 Type conversions with constructors

In Python it is quite common to enable passing values of all kinds of different types through the same argument of a function, using the dynamically typed nature of the language. Among other places, this is frequently allowed in single argument constructors, which type check the argument and try to convert it to newly created object accordingly. If conversion fails, the constructor raises an exception. Let's see a couple of examples, with the types mentioned so far:

```python
assert int('010') == 10
assert int('010', base=16) == 16
assert int(-3.5) == -3
assert int(b'010') == 10
assert float(1) == 1.0
assert float('1.2e8') == 120_000_000.0

assert bool(0) == False
assert bool(-3) == True
assert bool(0.0) == False
assert bool(0.01) == True
assert bool([]) == False
assert bool([0]) == True
assert bool(()) == False
assert bool((0,)) == True
assert bool(set()) == False
assert bool({0}) == True
assert bool({}) == False
assert bool({0: 0}) == True
assert bool('') == False
assert bool('0') == True
assert bool('false') == True
assert bool('False') == True

assert list((1, 2, 3)) == [1, 2, 3]
assert tuple([1, 2, 3]) == (1, 2, 3)
assert set([1, 2, 3, 2, 3, 4]) == {1, 2, 3, 4}
assert dict([(1, 2), (3, 4)]) == {1: 2, 3: 4}
assert list({1: 2, 3: 4}.items()) == [(1, 2), (3, 4)]
assert list({1: 2, 3: 4}.keys()) == [1, 3]
assert list({1: 2, 3: 4}.values()) == [2, 4]

assert str(1) == '1'
assert str(1.1) == '1.1'
assert str([1,2,3]) == '[1, 2, 3]'
l = [0, 1]
l.append(l)
assert str(l) == '[0, 1, [...]]'

assert list('abc') == ['a', 'b', 'c']
assert list(b'abc') == [0x61, 0x62, 0x63]
assert bytes([0x61, 0x62, 0x63]) == b'abc'

b = bytearray(b'abc')
```

```python
b[0] = 0x41
b[1:2] = b'BB'
b.append(0x42)
assert bytes(b) == b'ABBcB'
b.remove(0x42)
assert bytes(b) == b'ABcB'
```

3.4.5 A note on type annotations

Dynamic typing can become confusing on large code bases. How do you know exactly what type of argument a function can be called with. Technically of course you can call it with any type, but it may easily cause unexpected behavior, even in cases when you think you are doing the right thing. Of course type checking all your arguments is possible, but adds complexity to your code both measurable in number of lines and speed.

To make at least the programmers life somewhat easier in these scenarios, Python allows type annotations on variables and function signatures (i.e. arguments and return type). It is best to handle these type annotations as hints. The interpreter does not actually enforce passing the correct types, but some development tools might help you with warnings if you are doing something that contradicts the type annotations.

```python
def my_well_annotated_function(
    arg1: int,
    arg2: list[int],
    arg3: tuple[int, str, float],
) -> str:
    if arg1 == 2:
        b: int = 3
    else:
        b = 4
    return str(b) + arg3[1]

s = my_well_annotated_function(2, [1, 2, 3], (1, '2', 3.0))
```

3.5 Control flow: if-else, loops

It feels almost redundant to talk about `if` statements at this point, but I wanted to talk about them after going through the basic types. `if` does not require a boolean expression, it requires something that can be converted to a `bool`, which comes down to the concept of truthy and falsy values.

```python
a = 0
if a:
    a = 1
assert a == 0
```

For else-if, python has a shortened keyword `elif`:

```python
a = 0
if a:
    a = 1
```

```python
elif a + 1:
    a = 2
else:
    a = 3
assert a == 2
```

There is no switch-case or anything similar in Python, but you can always write an equivalent `if`-`elif`-`else` chain.

As most languages, Python has `while` and `for` loops.

```python
a = 10
while a:
    a -= 1
assert a == 0
```

A for loop goes over an iterable object, like a `list`:

```python
l1 = [1, 2, 3]
l2 = []
for x in l1:
    l2.append(x)
assert l2 == [1, 2, 3]
```

Some other iterable objects do not actually contain the items ahead of time, but their iterators retrieve them on the go one-by-one when the code consuming the iterator (like a `for` loop) requests for the next item. This concept is known as lazy-iterators. A good example of this are the iterable returned by the builtin `range` function. An easy way to inspect the behavior of finite iterators is to pass them to the constructor of `list`, which goes through them all the way, and collects the retrieved items:

```python
assert list(range(5)) == [0, 1, 2, 3, 4]
assert list(range(2, 5)) == [2, 3, 4]
assert list(range(5, 2)) == []
assert list(range(1, 7, 2)) == [1, 3, 5]  # similar to indexing
assert list(range(5, 2, -1)) == [5, 4, 3] #  of extended slices
```

There are multiple ways of creating such iterables, including generator expressions, generator functions, and making your own iterable types, all discussed later.

For now, let's see a couple of tricks with iterables that usually useful when working with `for` loops:

```python
l1 = [10, 11, 12, 13, 14]
l2 = [20, 21, 22, 23]

r = []
for i in range(len(l1)):
    r.append((i, l1[i]))
assert r == [(0, 10), (1, 11), (2, 12), (3, 13), (4, 14)]

r = []
for ix in enumerate(l1):
    r.append(ix)
assert r == [(0, 10), (1, 11), (2, 12), (3, 13), (4, 14)]

```

```python
r = []
for xy in zip(l1, l2):
    r.append(xy)
assert r == [(10, 20), (11, 21), (12, 22), (13, 23)]
assert len(r) == min(len(l1), len(l2))
```

You can use `break` and `continue` statements within both `while` and `for` loops. They will always interact with the inner-most loop around them.

```python
r = []
for i in range(10):
    r.append(i)
    if i > 3:
        break
    continue
    r.append(i) # skipped due to the continue above
assert r == [0, 1, 2, 3, 4]
```

An interesting feature that I have not seen in any other language, is that loops can have an additional `else` branch, which runs if the `loop` finished without breaking:

```python
p = 7
for i in range(2, int(p ** 0.5) + 1):
    if p % i == 0:
        break
else:
    print("p is prime")
```

3.6 Packing and unpacking

In the previous section on control flow we discussed loops with a touch to the topic of iterables. Iterables (and iterators) are quite useful when dealing with a lot of elements. Sometimes, however, we are dealing with a small constant number of elements still packaged in a collection, usually a `tuple`. It is usually not pleasant to access the elements one by one using their indices, so we extract them to separate variables:

```python
def my_function(t: tuple[int, str, float]) -> tuple[int, str, float]:
    a = t[0]
    b = t[1]
    c = t[2]
    ...
    t = (a, b, c)
    return t
```

Let's imagine that the above function somehow processes the three values extracted to `a`, `b`, and `c`, and returns them packaged back to a tuple. To make this easier we have a syntax to easily pack and unpack values:

```python
t = 1, 2, 3 # creates a tuple, just like with `()` around
a, b, c = t
```

This makes it quite natural to return multiple values from a function:

```python
def my_function2():
    return 1, 2, 3  # returns a tuple
a, b, c = my_function2()
```

It also works with lists and other iterable objects:

```python
a, b, c = [0, 1, 2]
assert (a, b, c) == (0, 1, 2)
a, b, c = range(3)
assert (a, b, c) == (0, 1, 2)
n, a, ii, v, e = 'naïve'
assert n == 'n' and ii == 'ï'
```

It enables easily swapping two variables:

```python
a, b = 1, 2
a, b = b, a
assert (a, b) == (2, 1)
```

You can use unpacking even in the head of `for` loops:

```python
l = [10, 11, 12, 13, 14]
r = []
for i, x in enumerate(l):
    r.append(f'[{i}]: {x}')
assert r == ['[0]: 10', '[1]: 11', '[2]: 12', '[3]: 13', '[4]: 14']
```

Note that if you try to unpack an iterable with a different number of elements, it will result in an exception.

3.7 Exception handling

In Python instead of throwing exceptions we `raise` them. Exceptions must all be descendants of the common superclass called `BaseException`. What happens if you try to `raise` something else (like `raise 1`)? Well, you will get an exception, but not the one you wanted. In Python all kinds of errors including referencing an undefined variable, division by zero, etc. will cause exceptions and can be handled with try-except blocks:

```python
def divide_by_zero(x):
    return x // 0

try:
    divide_by_zero(1)
except:
    print('as "excepted"')
```

To narrow down the exception type you want to handle you can provide a base type in the `except` statement. This way you can even have multiple `except` blocks to handle different types:

```python
try:
    1 // 0
except ValueError:
    print('this will not print')
except ZeroDivisionError:
    print('this will print')
except:
    print('this won\'t print either')
```

The last, general `except` branch is optional. If it is present, it catches all exceptions not matched by the other branches; otherwise such exceptions will remain uncaught and be sent up to the `except` branches of an outer `try` block, or if there are none, the caller of our function, and so on until it is caught or reaches the very top crashing the program. An `except` block can also take the exception object using the keyword `as`:

```python
try:
    1 // 0
except ZeroDivisionError as e:
    print(f'as "excepted": {e}') # prints `as "excepted": integer \
                                 # division or modulo by zero`
```

Instead of the `except` blocks or additionally to them, you can have a `finally` block, which will always runs after whatever happened in the `try` block. So `finally` will run even it there was an uncaught exception. This can be useful to release resources even if the program crashes.

If you have at least one `except` block, after them (but before the `finally` block if there's one), you can also have an `else` block, which works a bit similarly to the `else` block of loops. This `else` block runs if there were neither caught nor uncaught exceptions.

So, to sum it up:

```python
try:
    ... # runs until exception or all the way if there is none
except ZeroDivisionError as e:
    ... # runs if the given type of exception was raised in the try block
except ValueError as e:
    ... # runs if it catches an exception that was not
        # matched by previous `except` branches
except:
    ... # catches any exception that was not matched
        # by previous `except` branches
else:
    ... # runs if no exception happened in `try`
finally:
    ... # always runs, even if there was no general `except` block
        # and there was an uncaught exception handled somewhere else
        # or crashing the program

... # runs if there was no exception, or it was caught and handled
    # by one of the except blocks above
```

To raise an exception explicitly we can use the `raise` keyword. In an `expect` block you can use `raise` without any parameters to re-raise the currently exception, otherwise you must specify a kind of exception you want to raise. This can be either an instance of an exception type directly, or the name of an exception type alone, which then will be implicitly instantiated with default arguments.

```python
def function_working_with_int_arg_only(x):
    if not isinstance(x, int):
        raise ValueError # equivalent to `raise ValueError()`
```

Most builtin exception types can take an error message, which can be extracted where the exception is caught or is displayed if the exception is uncaught and crashes the application:

```python
def function_working_with_int_arg_only(x):
    if not isinstance(x, int):
        raise ValueError('something horrible passed to a function'
                         ' working with int arg only')
```

3.8 Advanced function signatures

In Python we can pass arguments to functions both in a given order without labeling them (positional arguments), or in an arbitrary order, but labeling each of them (keyword arguments).

```python
def sub(a, b):
    return a - b

assert sub(5, 3) == 2
assert sub(b = 3, a = 5) == 2
```

The methods can be even combined, by passing the first few parameters according their order, and passing others by their names:

```python
def example(a, b, c):
    ...

example(1, c=3, b=2)
```

Arguments can be made optional by providing default values for them:

```python
def example2(a, b=3, c=8):
    return a, b, c
assert example2(c=4, a=2) == (2, 3, 4)
assert example2(4, 5) == (4, 5, 8)
assert example2(5, c=2) == (5, 3, 2)
```

However, if an argument has a default value, all the subsequent arguments must also have a default value (except for keyword only arguments explained shortly).

3.8.1 Positional only and keyword only arguments

In a more complex function definition, you can also mark certain arguments as positional only and keyword only.

```python
def example3(a, /, b, c):
    return a, b, c
```

`/` is not an actual argument, it just marks the end of positional only arguments. This way `a` cannot be provided as a keyword argument.

```python
def example4(a, b, *, c):
    return a, b, c

# example4(4, 5, 6) is an invalid call
example4(4, 5, c=6) # must provide c as a keyword argument
```

Similarly, `*` here is not an actual argument either, it just marks the beginning of keyword only arguments.

Naturally the two restrictions can be combined:

```python
def example5(a, /, b, *, c):
    return a, b, c
assert example5(1, 2, c=3) == (1, 2, 3)
assert example5(1, c=3, b=2) == (1, 2, 3)
```

3.8.2 Variable number of arguments: `*args` and `**kwargs`

A function can be defined to take variable numbers of both positional and keyword arguments. Additional positional arguments can be taken with `*args` and additional keyword arguments can be taken with `**kwargs`. Note that `args` and `kwargs` technically could be named any other way, but usually these names are used. `args` is a tuple, `kwargs` is a dictionary.

```python
def example6(*args, **kwargs):
    print(args, kwargs)
```

The above `example6` function does not make any restrictions regarding its arguments: it can take both any number of positional arguments and any number of keyword arguments.

```python
example6(1,2,3,4,5,a=6,b=7,c=8) # prints `(1, 2, 3, 4, 5) \
                                # {'a': 6, 'b': 7, 'c': 8}`
```

In more complicated function signatures `*args` (if present) must be provided in place of `*` and `**kwargs` (if present) must be provided at the very end, after all kinds of other arguments. Note that values to other arguments explicitly listed in the function signature will not be included in either `args` or `kwargs`.

```python
def example7(a, /, b=None, *args, c=None, **kwargs):
    print(a, b, args, c, kwargs)

example7(1) # prints `1 None () None {}`
example7(1, 2) # prints `1 2 [] None {}`
example7(1, 2, 3) # prints `1 2 [3] None {}`
example7(1, 2, 3, 4) # prints `1 2 [3, 4] None {}`
example7(1, b=2) # prints `1 2 [3, 4] None {}`
# example7(1, 2, b=3) would error
#         `TypeError: example7() got multiple values for argument 'b'`
example7(1, 2, 3, 4, d=6, c=5, e=7)
                              # prints `1 2 (3, 4) 5 {'d': 6, 'e': 7}`
```

If keyword argument is passed with a matching name to positional only argument (like `a` above), it will be handled as two separate arguments. If `**kwargs` is present in the signature, the value provided to the keyword version will be added to `kwargs`, otherwise passing a keyword version will result in the exception.

```python
def example8(a, /):
    return a

# example8(a=1) would error
# example8(1, a=2) would error

example7(1, a=2) # prints `1 None [] None {'a': 2}`
```

This interesting behavior is useful because we can have some positional only arguments without taking away any possible names for keyword arguments.

3.8.3 Calling functions with a dynamic collection of arguments

Similarly to how a function signature can accept variable number of arguments, with `*args` and `**kwargs`, functions can also be called with a dynamically assembled set of arguments.

```python
def example9(a, b, c, d, e):
    print(a, b, c, d, e)

my_args = [1, 2, 3, 4, 5]
example9(*my_args) # prints `1 2 3 4 5`
my_args2 = {'e': 5, 'd': 4, 'c': 3}
example9(1, **my_args2, b=2) # prints `1 2 3 4 5`
```

This same syntax can also be used when defining tuples, lists, sets and dictionaries:

```python
my_tuple = (*my_args, 6, 7)
my_list = [0, *my_args, 6, 7]
my_set = {0, *my_args, 6, 7}
my_dict = {'a': 1, **my_args2, 'b': 2}
```

Using all this knowledge, we can write a totally useless universal wrapper function:

```python
def call_the_function(the_function, /, *args, **kwargs):
    return the_function(*args, **kwargs)

call_the_function(print, 1, 2, 3, sep=',') # prints `1,2,3`
```

You might notice that we have just passed a function to another function as an argument. In Python functions are objects just like everything else that has a name.

3.8.4 Lambdas

It is quite common that you need to define a really simple function that consists of evaluating a single expression. For this Python has a shorthand using the `lambda` keyword:

```python
my_lambda = lambda x: x * 2
assert my_lambda(3) == 6
```

This syntax is quite limited, as you really cannot add any additional statements to the logic represented by the lambda. Of course you can still call other functions, so this does not actually limit the underlying complexity.

Lambdas can still have all the kinds of advanced function signatures we reviewed in the previous sections:

```
example7_as_lambda = \
    lambda a, /, b=None, *args, c=None, **kwargs: print(a, b, args, c, kwargs)
```

3.9 Advanced expressions

Python has a few advanced syntaxes for expressions that might help you write more elegant code. Let's get to know some of them.

3.9.1 Conditional expression

Python has a syntax for inline conditional expressions, similar to the `?-:` operator of some other languages. Only the order of the three expressions is a bit different:

```
# value = value_if_true if condition else value_if_false
x = 2 if True else 3
assert x == 2
x = 2 if False else 3
assert x == 3
```

Just like in conventional `if` statements, the condition in the middle does not actually be a boolean, but rather anything that is convertible to `bool`.

3.9.2 List comprehension and generator expression

Its quite a common task to process an iterable, process its items and collect them to a new list or some other type of collection. Instead of calling methods like `map`, `filter` and `collect` on iterators, passing lambdas describing the required transformations and predicates, Python provides a different approach: list comprehension.

```
l1 = [1, 2, 3, 4, 5]
l2 = [2 * x for x in l1]
assert l2 == [2, 4, 6, 8, 10]
```

Multi-dimensional iterables can be processed flattened, by chaining multiple of these inner `for` clauses:

```
l1 = [[1, 2], [3, 4]]
l2 = [-x for l in l1 for x in l]
assert l2 == [-1, -2, -3, -4]
```

Note the difference from

```
l1 = [[1, 2], [3, 4]]
l2 = [[-x for x in l] for l in l1]
assert l2 == [[-1, -2], [-3, -4]]
```

You can also add filter conditions using the `if` keyword to these expressions:

```python
l1 = [[1, 2], [3], [4, 5]]
l2 = [-x for l in l1 if len(l) == 2 for x in l if x % 2 == 1]
assert l2 == [-1, -5]
```

You can also construct sets and dictionaries similarly:

```python
l = [1, 2, 3, 2, 1]
s = {-x for x in l}
assert s == {-1, -2, -3}
d = {x: i for i, x in enumerate(l)}
assert d == {1: 4, 2: 3, 3: 2}
```

Tuples cannot be in the very same manner, because `()` with such an expression is reserved for the so called generator expressions. A generator expression does not actually create a collection, only an iterator object that will lazily evaluate each item, when consumed.

```python
g = (2 ** i for i in range(5))
assert list(g) == [1, 2, 4, 8, 16]
assert list(g) == [] # can be consumed once
```

Generators are practical when dealing with a huge number of items that you do not want to simultaneously store in the memory.

3.10 Coming soon

More advanced topics coming soon!

What is planned:

- Imports and project structure
- Creating your own types: classes
  - Magic functions
  - Inheritance
  - Class level variables
  - Static methods
  - Properties
- Writing your own decorators
- Resource management with `with-as`
- Generator functions
- Dynamic metaprogramming