Einführung in die Programmierung mit Python

Heinz Hofbauer

WS 2022 *

Contents

*Version from November 8, 2022

Lecture

Part of this course is built as a lecture (this part) and will be structured somewhat like a tutorial.

Lecture notes Will be updated on the EIPMP21 Page
Contact me at hofbauer@cs.sbg.ac.at
Find me in person at Raum 0.27 Jakob-Haringer Str. 2

- If you have questions during the lecture part feel free to ask.
- If you have questions regarding an assignment it is easiest to send an email.

Times and places are kept up to date on the webpage of the course, see PLUS Online for a current link.

The **Offline** course is usually held in the Christian Doppler Hörsaal

Final Mark, Tests and Assignments

To conclude the course positively the following criteria has to be fullfilled:

1. Pass the test.
2. Obtain a positiv mark on the assignments.

The overall mark will be purely based on the mark for the assignments, under the precondition that the test is passed.

If either the test is not passed or the assignment mark is negative then the overall mark will be negative (5).

Test

- The test will be repeatable and is more or less for self reflection.
  Still it is enforced and must be passed (75% score or higher).
- Will be made available in Blackboard, duration during which it must be passed will also be posted.

Final Mark Assignments

- Key based on a percentage of maximum number of points:

Percentage of Points	0-60	>60	>70	>80	>90
Mark	5	4	3	2	1

- A single Assignment is usually 1 Point unless specified otherwise.
- Assignment 0 and Assignment 0 with git are 1 point in total (0.5 each).

- Longer assignments may be split into parts which can be individually solved
  - So as to give partial points for a partial solution.
  - These can be grouped in one assignment in artemis or invidvidual assignments.
- **Important**: If two or more solutions are too close (plagiarism) **all of them** will count as incorrect.
- Assignments should be submitted via the Artemis server
  - Assignments should automatically be evaluated.
  - However, this can be overridden on manual evaluation or plagiarism!
  - If you find bugs in the automatic evaluation please let me know.

Assignments and Results

The plan with assignments is as follows:

- At the beginning of the lecture we will talk about the old assignment.
  - I will provide a solution for some assignments.
  - Ask questions if something is still unclear.
- We will talk about the new assignments at the end of the lecture.
  - *If something is unclear ask* so you can efficiently work on the assignment.
  - The due date will be set in the Artemis system (look there).
  - The online systems used for turning the assignments in should provide feedback.
  - Assignments (and the lecture) are by nature building on one another, so do not neglect earlier assignments.
- Points and Test results.
  - Points for the tests and assignments will be posted on the Blackboard system.

Getting Python

General:

- We use Python 3
- Get it here: https://www.python.org/downloads
- Documentation: https://docs.python.org/3
- Some OS specific install instructions:
  - Ubuntu: `apt install python3`
  - Ubuntu install the idle editor: `apt install idle`
  - Windows 10: In the Microsoft store
  - MacOSX: Download the installer from the python website (should come with idle)

Getting an Editor

Basically use whatever text editor you like, the standard python input files are simple text files. The files can be UTF-8 encoded but do not need to be.

- Python comes with a default editor and interface, in Ubuntu it is split (idle package):
  - `idle`
- Other editors frequently recommended:
  - Notepad++ (win)
  - vim (linux, `apt-get install vim`) hard to master but extremely powerful
  - emacs (linux, `apt-get install emacs`) hard to master but extremely powerful
  - others in linux: gedit, nano
- IDEs (integrated development environment)
  These are more complex but also much more powerful
  - With proper configuration vim and emacs also fall into this category.
  - Eric
  - pydev
  - Visual Studio, should be free as *Community Edition*
  - PyCharm There should be a free student version

Compiler vs. Interpreter

We write programs in a high level language (abstracted from hardware). In essence we use an editor to generate **source code**. Source code usually denotes the programming language in the extension of the file, for python the files end in `.py`, for C the extension is `.c` for java we have `.java` and so on.

A **compiler** translates a high level language source code to a specific hardware (and usually operating system). It generates an **executable file**, compilation is done once and then can be executed multiple times.

An **interpreter** reads source code and interprets it by executing the given statements one at a time. This is done every time the code is executed.

Usually interpreters generate an in-between file usually called **bytecode**. This bytecode is like an executable for a **virtual hardware**. This is then run in a **virtual machine** emulating this virtual hardware.

Python generates `.pyc` files, java generates `.class` files.

What is programming?

Basically the task is to tell something with no intelligence but high capability what to do.

There is no intelligence on part of the computer, it is very literal. So every sort of thought has to come from the programmer.

However, due to the use of a high level language some of this is abstracted.

Example: `3 + 4 * 2`

- We know this is 11.
- The hardware when presented with the sequence would calculate `3 + 4 = 7` then `7 * 2 = 14`. Because it is written in that order.
- High level language know the concept of operator precedence (multiplication and division before addition and subtraction) so a high level language like python would interpret that correctly and calculate 11.

Try it out

Try it out in idle.

```
>>> 3+4*2
11
```

- `>>>` is the prompt, the rest of the line (`3+4*2`) is my input.
- Following that is the output produced by the interpreter.

If you start idle you have an interactive interpreter at hand.

Note: If you copy something verbatim from this interpreter interface you do not have a valid python program. Specifically the `>>>` prompt will cause a syntax error!

Operators and Precedence

Mathematical operation in python are in the following precedence order (like with math you can use parentheses `()` to group):

Operation	What is it
`**`	Exponentiation (raise to the power)
`~ + -`	Complement, unary plus and minus (as in -1 and +1)
`* / % //`	Multiply, divide, modulo and floor division
`+ -`	Addition and subtraction
`>> <<`	Right and left bitwise shift
`&`	Bitwise 'AND'
`^`	Bitwise exclusive `OR' and regular`OR'
`<= < > >=`	Comparison operators
`<> == !=`	Equality operators
`= %= /= //= -= += *= **=`	Assignment operators
`is is not`	Identity operators
`in not in`	Membership operators
`not or and`	Logical operators

Variables

The area A of a rectangle with width w and height h is $A = w * h$.

Consider what w or h are. They are variables, placeholder for a value. They allow us to talk about the area, which changes depending on the actual value of w and h independent of the actual value.

Variables are obviously useful, and also exist in python.

```
>>> w=7
>>> h=10
>>> w*h
70
```

This can also be assigned to a variable.

```
>>> A=w*h
>>> A
70
```

What can be a variable?

Obviously not a good idea (and not allowed in python):

```
>>> 7=3
SyntaxError: can't assign to literal
```

What can we use then as variables:

- Must start with underscore (_) or letter
- Rest can consist of numbers, letters and underscore.
  - `_test`
  - `w`
  - `variable1`
- Variables are case sensitive! The following are different variables
  - `variable`
  - `Variable`
  - `VARIABLE`
- Variables have a name
- Variables are a placeholder for a value (if not used for assignment)
- Variables can be assigned a value (by being followed by a `=`)

Think of a number:

```
>>> number1 = 17
>>> number2 = 3.1415
>>> number3 = "a number"
```

- Variables also can be undefined (meaning they don't exist)

```
>>> number4
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    number4
NameError: name 'number4' is not defined
```

In addition to that variables can also have no data. The value expression for this is `None`.

```
>>> number5 = None
>>> number5
```

There is no output, but there also is no error. This is often used to set a variable to an error state.

But wait a moment, `None` is a valid variable name, can I use it?

```
>>> None = 4
SyntaxError: can't assign to keyword
```

No you can not as this would make programming extremely complicated (and error prone).

Words like this are called **keywords**, they have a special meaning in the programming language.

Keywords

These are keywords in python and can not be used as variable names.

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Variables cont.

What is a value?
A value has a type, e.g., it is an integer, a real number or letters. But for the most part python does not care but uses duck typing.

> If it walks like a duck and it quacks like a duck, then it must be a duck.

- 4 (integer) times 3 (integer)

  ```
  >>> 4*3
  12
  ```

  results in an integer, ok.

- 17 (integer) times 3.1415 (real)

  ```
  >>> 17*3.1415
  53.4055
  ```

  becomes a real number, makes sense.

- “number” (word) times 3 (integer), now what?

  ```
  >>> "number" * 3
  'numbernumbernumber'
  ```

  Is literlly the word “number” taken three times.

- “number” (word) divided by 3 (integer), what could that be?

  ```
  >>> "number" / 3
  Traceback (most recent call last):
    File "<pyshell#1>", line 1, in <module>
      "number" / 3
  ```

```
TypeError: unsupported operand type(s) for /:
        'str' and 'int'
```

Python does not know either.

- "3" (a word with the letter 3) times 3 (integer):

```
>>> "3" * 3
'333'
```

Again literally the word "3" taken three times.

So in essence python tries to behave like it is intelligent but isn't.

Is there a way we can deal with this explicitly?

The answer is yes and there are lots of types in python. But more on this later, for now be aware that python tries to do the right thing. We as programmers have to help it along though.

Functions

So currently we basically have a, somewhat quirky, pocket calculator on our hands.

What differentiates programming from a pocket calculator is that we can write and reuse code.

Think of this in terms of mathematical functions to get the basic idea.

- $sin(x)$ takes a variable x and calculates the sinus of it.
- $f(x, y) = \frac{x}{y}$ takes variables x and y and divides one by the other.

Python can also do this and the way we write this (the **syntax**) is as follows:

```
def f(x , y):
    result = x / y
    return result
```


```
def f(x , y):
    result = x / y
    return result
```


- `def` is a keyword specifying that this is a function declaration.
- It is followed by a function name, `f` in our case.
- Then there is a list of variables, separated by a comma and encased in parenthesis: `(x, y)`
- After that the colon `:` specifies that the **declaration** is finished.

But what does the function do?


```
def f(x , y):
    result = x / y
    return result
```


Following the declaration is the **function body**, which tells python what do to when it see `f(x,y)`.

- This is a **block** of instructions.
- A block is a grouping of instructions which belong together.

- Python specifies a block by indentation (either with space or tabulator):
  - Take care here the indentation *must be the same for each line*
  - Do not mix spaces and tabulator, do not change the number of tabs or spaces

Finally the function returns a result with the `return` keyword.

Some examples:

```
>>> def f(x, y):
...     result = x / y
...     return result
>>> f(1,2)
0.5
```

But this will not work:

```
>>> def f(x, y):
...     result = x / y
...   return result
SyntaxError: unindent does not match any outer indentation
level
```

Function names and variable names have the same naming rules and live in the same **namespace** (more on that later).

```
>>> def f(x, y):
...     result = x / y
...     return result
>>> f(1,2)
0.5
>>> f = 5
>>> f(1,2)
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    f(1,2)
TypeError: 'int' object is not callable
```

Python and Files

So far so good, so we can declare functions and reuse them. But we still only have a pocket calculator where we do have to declare the function every time we want to use it.

But we can store the source code in a files. A plain text files (which can be utf-8 encoded).

On the command line we can then execute it with the python interpreter to do our calculations.

- Create a file called `test.py` with the following contents.

  ```
  def f(x, y):
      return x / y

  f(1,2)
  ```

- run `python test.py` in the console (unix) or `cmd` (windows) or directly in idle (`F5`).

You will notice nothing happens. Specifically there is no output, that is because the interactive shell does report back an output in proper program we as the programmers control the output.

There is a build in function to use: `print()` (guess what was used as output medium before there were displays).

So lets amend the program and:

```
def f(x, y):
    return x / y

print( f(1,2) )
```

`F5` and we get:

```
0.5
```

Order of Execution

Note that the file is read and starting from the top it will be executed one line at a time.

This if we want to use function before we declare it, that will not work. Example:

```
print( f(1,2) )

def f(x, y):
    return x / y
```

`F5` and we get:

```
Traceback (most recent call last):
  File ".../test.py", line 2, in <module>
    print(f(1,2))
NameError: name 'f' is not defined
```

Python and Files cont.

So we have the code saved in a file, this code can be reused! It was stored in a files called `test.py` and other files (in the same directory) can load it.

The command for importing other codes and modules is

`import MODULE`

The import statement combines two tasks:

1. It searches for the module to import.
2. If one is found the module is imported (basically the files is executed).

Function from the modules can be used but they exist in a different namespace (that of the module).

Note that the search will be done in certain system paths (installed by package manager or windows store) as well as the local directory (as part of a project).

```
>>> import test
0.5

>>> f(1,2)
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
```

```
    f(1,2)
NameError: name 'f' is not defined
```

To access the namespace of the module we prefix `module.` :

```
>>> test.f(1,3)
0.33333333333333333
```

There are two other ways to import

1. Pull some definitions into the current namespace:

   `from MODULE import FUNCTION1, FUNCTION2 ...`

   ```
   >>> from test import f
   >>> f(1,4)
   0.25
   ```

   A shorthand for importing everything into the current namespace is:

   `from test import *`

   Please don't do this.

2. Rename the namespace (mostly used to shorten long module names):

   `import MODULES as SOMETHINGELSE`

   ```
   >>> import test as t
   >>> t.f(1,5)
   0.2
   ```

Recall that the import had an output, why is that?

```
>>> import test
0.5
```

Remember that the import executes the imported file (so the definitions are correctly set). Well our file also had a print statement in it, recall:

```
def f(x, y):
    return x / y

print( f(1,2) ) <--- The print statement
```

This is also executed!

So this is unwanted behavior, can it be prevented? Obviously the answer is yes.

Python sets a special variable `__name__` depending on how we execute the file.

- If the files is executed it will be set to `__main__`.
- If it is loaded as a module it will be set to the filename with the `.py` extension.

```
def f(x, y):
    return x / y

print( __name__ )
```

Import it and run it and see what happens.

Conditional

How can we make use of this? The answer is conditional execution, that is some part of the code is only executed **IF** a certain condition is met:

```
def f(x,y):
    return x/y

if __name__ == "__main__":
    print( f(1,2) )
```

The syntax looks like this:

```
if CONDITION:
    BLOCK
else:
    BLOCK
```

As can be seen in our example the else part is optional.

The `CONDITION` is something that evaluates to a truth value, either `False` or `True`.

- This can be any statement python can evaluate (for example a function which returns `True` or `False`).
- But the most common are comparison operators which compares two operands

  ```
  LEFT_OPERAND Operator RIGHT_OPERAND
  ```

- If the `CONDITION` evaluates to `True` the first block (below the `if`) is executed.
- If it is `False` the `else` block is executed. If there is no `else` block nothing is executed.

For the following assume that `a=1 and b=2`. If the result is not `True` it is `False`.

Operator	Description (Is `True` if …)	Example
`==`	… the two operands are equal	`a == b` is `False`
`!=`	… the two operators are *not* equal	`a != b` is `True`
`>`	… the left operand is greature than the right operand	`a > b` is `False`
`<`	… the right operand is greater than the left operand	`a < b` is `True`
`>=`	… the left operand is greater or equal to the right operand	`a >= b` is `False`
`<=`	… the right operand is greater or equal to the left operand	`a <= b` is `True`

- **NOTE**: `=` is used as assignment of a value to a variable, `==` is the equality operator!

Examples with Numbers:

```
>>> 1 == 3
False
>>> 1 <= 1.0
True
>>> 1 != 0
True
>>> -10 > 5
False
```

Examples with Text:

```
>>> __name__ == "__main__"
True
>>> "a" < "b"
True
>>> "A" < "b"
True
>>> "a" < "B"
False
```

So basic alphabetic ordering works with capitals before lower caps:

- ABC…XYZabc…xyz

Some more examples

```
>>> True == False
False
>>> True < False
False
>>> True > False
True
```

OK so apparently there is also an order to truth and `False` is less than `True` (good ordering I think).

```
>>> "," < "a"
True
>>> 1 == "1"
False
```

Wait what? 1 is not "1"?

Types

Remember duck typing? Python tries to do the right thing but sometimes we have to help.

To help we have to know what is going on, so it's often (almost always really) beneficial to know what type a variable is.

Python has a lot more types, which we will touch upon later but the above list is pretty common over all programming languages:

Type	Description
bool	**Boolean** types hold a truth value (`True` or `False`) and are use for boolean logic
int	**Integer** variables hold an integer number ($\mathbb{Z}$), that is a number without a fraction
float	**Floating point mumbers** hold a real number ($\mathbb{R}$) so a number with a decimal point
string	**Strings** are a sequences of characters.

What do we do if a variable comes along and we want to know it's type?

We use the `type()` function, which in this basic application takes an argument and returns the type.

```
>>> s="asd"
>>> b=True
```

```
>>> i=17
>>> f=123.123
>>> type(s)
<class 'str'>
>>> type(b)
<class 'bool'>
>>> type(i)
<class 'int'>
>>> type(f)
<class 'float'>
```

Can directly applied to values too (to see what the variable will be)

```
>>> type ("string")
<class 'str'>
```

Can also be applied to functions, expressions or similar. Those are first evaluated then the type taken.

The expression is an comparison so the result is a bool (truth value).

```
>>> type( i == f )
<class 'bool'>
```

The return value of the function is a floating point number.

```
>>> import test
>>> type( test.f(1,2) )
<class 'float'>
```

Can we convert the types?

Yes! Every type has a function which takes an argument and tries to convert the argument to it's type.

```
>>> int( 123 )
123
>>> int( "123" )
123
>>> int(123.123)
123
```

Looks pretty sensible so far, but beware:

```
>>> int(123.923)
123
```

Floating point to integer conversion does not round, it simply cuts of trailing decimals.

Conversion to floating point number work in a similar fashion:

```
>>> float( 123 )
123.0
>>> float( 123.123 )
123.123
>>> float( "123.123" )
123.123
```

Float conversion knows about scientific notation shorthand 1.23123×10^2 is represented as 1.23123e2.

```
>>> float( "1.23123e2")
123.123
```

Also note that if there is a decimal point its a float:

```
>>> type( 2.)
<class 'float'>
```

Strings also work as expected:

```
>>> str(123)
'123'
```

Strings are denoted by opening and closing single (') or double (").

```
message="Hello World"
message2='Hello World'
```

Strings can also be denoted by opening and closing triple (""" or ''') quotes. In this case the line breaks can be used in the string.

```
message3 = """Hello
World"""
message4 = '''Hello
World'''
```

What if we want to use ' or " in the string? Well use the other one to open/close the string.

```
whatif = 'This is a double quote: "'
```

What if we want to use both? There are two answers:

- Chain strings, that is write two string one after the other an python will combine them:

  ```
  >>> whatif2 = "These quotes can denote strings:" "'" '"'
  >>> print( whatif2 )
  These quotes can denote strings:'"
  ```

- Use escapes, python uses the character \ to give the next character a special meaning: some of these are \" is replaced with " and \' is replaced with ' in a string.

  ```
  whatif3 = "These quotes can denote a string: \"\'"
  print( whatif3 )
  These quotes can denote a string: "'
  ```

There are a number of other such escape sequences, the most obvious being the on which let's us write a literal \, for which the escape sequence is \\.

Another important on is \n which is turned into a newline (a line break):

```
>>> print("To open close a string use: \'\""
>>>       "or triple quotes\nAs an escape character use \\")
To open close a string use: '" or triple quotes
As an escape character use \
```

There are others escape sequences, but these are the most useful.

What is also often useful for working with strings is that addition does work for string.

String addition is concatenation of strings:

```
>>> "Hello" + "World"
'HelloWorld'
```

Works only with string and string though

```
>>> "Hello" + 3
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    "Hello" + 3
TypeError: must be str, not int
>>> "Hello" + str(3)
'Hello3'
```

Note that space are not introduced during the addition!

So booleans, simply `True` and `False`, how hard can it be?

```
>>> bool(True)
True
>>> bool(123)
True
>>> bool(0)
False
>>> bool(-123)
True
>>> bool( 123.123)
True
>>> bool("True")
True
>>> bool("False")
True
```

So what is going on here?

The convention is:

- If it has a truth value use that (`True` and `False`).
- If it is a number:
  - 0 is `False`
  - All others are `True`
- If it is not a number use it's length and treat it as a number.

An empty string (length is zero):

```
>>> bool("")
False
>>> bool("Not Empty")
True
```

This holds true for everything (`list`, `dict` and classes too). More on some of that later.

There is a function to check length:

`` `len( ARGUMENT )` ``

which returns the length of it's argument.

```
>>> len( "Hello" )
5
>>> len( "" )
0
```

Not everything has a length though.

```
>>> len( 123 )
Traceback (most recent call last):
  File "<pyshell#156>", line 1, in <module>
    len( 123 )
TypeError: object of type 'int' has no len()
```

It is not immediately obvious why this distinction is made, but what `len()` does is to count the number of objects in it's argument.

- Remember that a string is a sequence of characters.
- A number is simply a number.

This is mostly based on the representation of information in computers, again more on that later.

For now let us return to what prompted this excursion, the `if` statement and conditionals.

Boolean logic

Conditionals use boolean logic and the operators on boolean logic are implemented in python too.

- `A` **`and`** `B` is `True` if both A and B are `True` and `False` otherwise.
- `A` **`or`** `B` is `True` if A or B are `True` and `False` otherwise.
- **`not`** `A` reverses the truth value of A (`True` becomes `False` and vice versa).

So pretty obvious what it does, here is the interesting part: `A` and `B` are also conditionals.

An example would be something like this:

```
day_of_week == "Monday" and ( weather == "sunny" or weather == "cloudy" )
```

Conditionals for error handling

We already saw one use for conditionals but there are a lot of uses for them.

A very typical one is error handling, recall our function `f`:

```
>>> def f(x, y):
        return x/y

>>> f(5,0)
Traceback (most recent call last):
  File "<pyshell#180>", line 1, in <module>
    f(5,0)
  File "<pyshell#179>", line 2, in f
    return x/y
ZeroDivisionError: division by zero
```

This will halt our program (actually it raises an **exception** which is another type of error handling we will discuss later).

How can we deal with this? Consider the following

```
>>> def f(x,y):
        return_value = None
        if y != 0:
```

```
            return_value = x/y
        return return_value

>>> print( f(5,0) )
```

- We set an undefined (`None`) return value.
- If the division is valid (`y != 0`) then we store the correct value in the return value.
- Then we return the value.

As a side note, the print is necessary since the interpreter would not otherwise show the `None` result.

Cascading `if`s and `elif`

What if we have more than two cases, like in the weather example above, something like this.

```
if weather == "sunny":
    BLOCK
else:
    if weather == "cloudy":
        BLOCK
    else:
        if weather == "stormy":
            BLOCK
        else:
            BLOCK
```

This is ugly and hard to read, mainly due to the increasing indentation due to blocks.

The solution in python is called `elif`, which combined else and if into one statement.

This code is functionally the same as the above code but much nicer to read.

```
if weather == "sunny":
    BLOCK
elif weather == "cloudy":
    BLOCK
elif weather == "stormy":
    BLOCK
else:
    BLOCK
```

There can be any number of `elif` statements, there can also be a final (optional) else statement. But remember a given `elif` is only tested if all that came before failed, ordering is important here!

As an example let's output the temperature in words depending on the temperature in degree, like so.

```
if temperature < 10:
    print("It's cold")
elif temperature < 20:
    print("It's not warm")
elif temperature > 400:
    print("I'm burning")
elif temperature > 30:
    print("It's hot")
else:
    print("This is fine")
```

Try it out and really think about what range temperature can have at a given `elif` statement. As an example, at the first `elif` statement the temperature will be at least 10°C.

While Loop

So far we have a very linear program, it starts at the top and goes through once.

We can prevent some parts of the code from running with the help of conditionals.

What if we want to do something more than once, we use **loops**.

Example: We want to print all fractions of the form $1/N$ which are larger than `0.1`.

```
N=1
while 1/N > 0.1:
    print(1/N)
    N = N + 1
```

What does the `while` keyword do?

It works like this:

```
while CONDITION:
    BLOCK
```

while the condition evaluates to `True` the block will be executed once, then the CONDITION is again evaluated.

IMPORTANT:

- If the CONDITION is `True` run the BLOCK.
- Repeat this until the CONDITION is `False`.
- Only the BLOCK is executed so the result of the CONDITION can only change because of something in the BLOCK!
- If the CONDITION does not change this loop will be executed forever (**infinite loop**).

Let us look at the basic components of a loop:

1. Initialization

   ```
   N = 1
   ```

   This initializes variables used in the condition and inside the loop.

2. Conditional part, i.e. when or how long to run a loop.

   ```
   while 1/N > 0.1:
   ```

3. The content of the loop, that is what the loop actually does.

   ```
   print( 1/N )
   ```

4. Update of the conditional, strictly speaking this is of course part of the content of the loop:

   ```
   N = N + 1
   ```

   We increase N by one.

Input and Output

We already know the basic usage of `print()` for outputing information. Its companion is `input()` which lets us read input from a user.

`input( [prompt] )` Prints the prompt and waits for input from the user by reading a line and returns it as a string without the newline. The enter key will input a newline.

Example:

```
name = input( "What is your name? " )
print( "Hello " + name )
```

The program does not know what information was entered so it returns a string (as the is the most generic variant).

An example, pass and commenting

As an example let us write a program which lets the user enter a number and print all fractions of the form $1/N$ larger than that fraction.

Note: For larger programs it is often useful to break the program down into smaller parts and tackle them individually.

So in our case we need an input from the user and then output the fractions greater than that number.

```
if __name__ == "__main__":
    threshold = get_user_input()
    print_fractions( threshold )
```

However if we run that now (to test) it will complain that the functions do not yet exist.

So let us create the functions and simply do not do anything in them yet.

For functions with a return value it makes sense to return something, either `None` but that might break things later down, or a value which lets us test the program.

```
def get_user_input():
    return 0.1

def print_fractions( threshold ):
    pass

if __name__ == "__main__":
    threshold = get_user_input()
    print_fractions( threshold )
```

Note the keyword `pass`! A block can not be empty but the `pass` allows us to create a block with an instruction (`pass`) which simply does nothing.

This is a valid program and we can run it!

Also the program is getting longer and we have to remember stuff, like changing the 0.1 in `get_user_input` to an actual value. Usually this is done by adding comments into the code.

- Comments in python start with a `#` character.
  - The `#` character is considered the start (and part) of a comment

  - Everything following it (until the end of the line) is considered a comment.
- This is information that the interpreter ignores.

```
def get_user_input():
    # TODO
    return 0.1 # change this

def print_fractions( threshold ):
    # TODO
    pass
```

A comment is not a valid block, because the interpreter ignores it.

So let's carry on, the print_fractions part is actually easy, we already did this.

Lets copy that code and adjust to use the threshold:

```
def print_fractions( threshold ):
    N = 1

    # change here from 0.1 to threshold
    while 1/N > threshold:
        print( 1/N )
        N = N + 1
```

Also note that there is no return directive, the function does not return anything. In this case the return value defaults to `None`.

So on to the final part, the user input.

```
def get_user_input():
    threshold =  input("Give a threshold please: ")

    #this is a string but we want a floating point threshold
    threshold = float( threshold )

    return threshold #return the correct value
```

Maybe we should also tell the user what the program does so the input makes more sense. Let's amend the main part of the program.

```
if __name__ == "__main__":
    print("Will print all fractions 1/N above a given threshold.")
    threshold = get_user_input()
    print_fractions( threshold )
```

The final program is then:

```
def get_user_input():
    threshold =  input("Give a threshold please: ")
    #this is a string but we want a floating point threshold
    threshold = float( threshold )

    return threshold #return the correct value

def print_fractions( threshold ):
    N = 1

    # change here from 0.1 to threshold
    while 1/N > threshold:
```

```
        print( 1/N )
        N = N + 1

if __name__ == "__main__":
    print("Will print all fractions 1/N above a given threshold.")
    threshold = get_user_input()
    print_fractions( threshold )
```

Let's run it:

```
========== OUTPUT ==========
Will print all fractions 1/N above a given threshold.
Give a threshold please: 0.1
1.0
0.5
0.3333333333333333
0.25
0.2
0.16666666666666666
0.14285714285714285
0.125
0.1111111111111111
>>>
========== OUTPUT ==========
Will print all fractions 1/N above a given threshold.
Give a threshold please: 1
```

The first we know already, in the second case there is simply no output. So working as intended?

What if we want to fuck with it?

```
========== OUTPUT ==========
Will print all fractions 1/N above a given threshold.
Give a threshold please: threshold
Traceback (most recent call last):
  File "/tmp/ourprog.py", line 16, in <module>
    threshold = get_user_input()
  File "/tmp/ourprog.py", line 4, in get_user_input
    threshold = float( threshold )
ValueError: could not convert string to float: 'threshold'
```

Can we construct a conditional here to prevent the error? That would require quite a complicated or long if statement or some other form of working with strings.

This does exist and such expressions are called **regular expressions**, we will not cover that.

Error handling and exceptions

The pythonic way for this to just try it and if it fails deal with the error.

This is quite a frequent approach for other programming languages as well.

So first lets look at what the error does:

1. An **exception** is raised with a *type* and *description*.
2. Normal execution is stopped until some part of the code deals with this exception.
3. If the exception was handled before the program ends the code is executed again from then on.
4. If the exception was not handled before the program ends an error statement is printed as well as where it happened.

The **traceback** gives us a trace back to where the error happened.

```
Traceback (most recent call last):
  File "/tmp/ourprog.py", line 16, in <module>
    threshold = get_user_input()
  File "/tmp/ourprog.py", line 4, in get_user_input
    threshold = float( threshold )
```

It is a sequence of where the error is and what was executed on that line:

1. First in `/tmp/ourprog.py` on line 16 the code
   `threshold = get_user_input()`
   was executed.
2. This leads to file `/tmp/ourprog.py` on line 4 and the execution of the line
   `threshold = float( threshold )`
3. Here the trace ends, so this is where the error occurred.

What was the error then, this is the last line:

```
ValueError: could not convert string to float: 'threshold'
```

It tells us the following: The type followed by a description

Type `ValueError` which means: *an operation or function receives an argument that has the right type but an inappropriate value* (from standard documentation).
Error Message `could not convert string to float: 'threshold'`

So the function received a string, which is OK, but the string contained the value 'threshold' which is of course not a number.

Note: If an exception occurs, check if you made a mistake during programming and should fix it to *prevent* the error, or the code is correct and you need to *handle* the error.

The code for handling exceptions is pretty straightforward:

```
try:
    #A block which might throw an exception
except Exception_type_we_want_to_catch:
    #Do something when an exception was thrown.
else:
    #Do something wenn no exception was thrown. This is optional
```

In our case the offending line was the conversion from float to string, the exception we want to handle is a `ValueError`.

```
try:
    threshold = float( threshold )
except ValueError:
    # But what do we do here?
```

The question of how to handle an error is kind of an important one. And as programmers we have to decide what to do, there is often not a *right* answer.

We could for example say:

```
try:
    threshold = float( threshold )
except ValueError:
```

```
    threshold = 1
```

And simply substitute a default. The program will continue to work. But the user might be annoyed, especially since we did say what happened.

We can also use the except form where we get the error message:

```
except EXCEPTION as exception_object:
    #exception block
```

The exception_object contains a lot of information about the exception but what we need is the error message. We can simply print the object and it will give the error message.

```
try:
    threshold = float( threshold )
except ValueError as msg:
    print( msg )
    print( "Assuming threshold = 1" )
    threshold = 1
```

So we do the same as before but also tell the user what is going on.

But what if it was a mistake by the user? Maybe we should ask again.

```
def get_user_input():
    # initialize the loop
    threshold = None

    # while we do not have a threshold
    while threshold == None:
        # Ask for input
        threshold = input("Give a threshold please: ")
        try:
            # try to convert (and update loop condition)
            threshold = float( threshold )
        except ValueError as msg:
            # if it failed, tell the user and update the loop condition
            print(msg)
            threshold = None

    #return the value
    return threshold
```

Another alternative is to simply put the whole get_user_input into a try block like so:

```
if __name__ == "__main__":
    print("Will print all fractions 1/N above a given threshold.")
    try:
        threshold = get_user_input()
    except Exception:
        pass
    else:
        print_fractions( threshold )
```

Note: Exception is a base for all exceptions and will catch them all. Usually do not do this!

What the code does is the following:

1. Try to get user input.
2. If anything goes wrong do nothing.

3. If we got an input print the fractions.

If can catch a single exception or all exception what should we do if there can be two different error types?

We can simply repeat the except block for another exception:

```
try:
    problem_function()
except ValueError:
    # handle ValueErrors
except ZeroDivisionError:
    # handle division by zero
```

So error handling is quite useful, how can we use exceptions for error signaling?

```
raise Exception("message")
```

`Exception` again is generic, use as specific an exception as possible!

More loops: break and continue

Some more on loops. If you take the last example we had to signal the loop to go on or stop with the threshold value.

This is often bothersome and there are two commands which can help to control the loop:

```
break
```

will end a loop when it is executed, and

```
continue
```

will immediately restart the loop and reevaluate the condition.

Use what is most readable in code and makes the most sense! Otherwise code becomes hard to read and overly complicated.

Consider this

```
def get_user_input():
    #loop forever, gives a hint to look for a break statement
    while True:
        threshold = input("Give a threshold please: ")
        try:
            threshold = float( threshold )
        except ValueError as msg:
            # if it failed, tell the user
            print(msg)
        else:
            #success, so stop the loop
            break

    return threshold
```

This is also quite clear and we do not have to worry about the state of the `threshold` variable, which in turn makes to code less error prone.

Variables, Scope and Namespaces

Recall the program and look at the variable names, that is all the same variable `threshold` is it not?

```
def get_user_input():
    threshold = input("Give a threshold please: ")
    threshold = float( threshold )
    return threshold

def print_fractions( threshold ):
    N = 1
    while 1/N > threshold:
        print( 1/N )
        N = N + 1

if __name__ == "__main__":
    print("Will print all fractions 1/N above a given threshold.")
    threshold = get_user_input()
    print_fractions( threshold )
```

So why do we even have to bother with return and function arguments?

The answer is of course because it is not the same variable.

Imagine every variable defined in a program would be globally readable and changeable. If there every is a strange error because of a wrong variable value or type it would be almost impossible to find (at least for longer programs).

Therefore there exists the concept of **namespaces**, we learned a bit about this before with importing other modules.

There are different namespaces in a program which exist in a hierarchy.

1. The builtin namespace
2. The global namespace
3. The function namespace

They are searched in order of most recent first (so first the function namespace, then the global namespace then the builtin namespace).

The **builtin namespace** is generated for every python program and populated with a number of functions that are always useful, some like print we know already:

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

Then comes the **global namespace**.

- Contains variables defined outside of functions.
- Import also creates variables and objects here.
- Function names are also in this namespace.

These overwrite builtin variables and functions, since on name conflicts these will be found first!

```
>>> print = 7
>>> print("asdf")
Traceback (most recent call last):
  File "<pyshell#195>", line 1, in <module>
    print("asdf")
TypeError: 'int' object is not callable
```

`print` is now an integer!

What can we do in this case?

Or more generally can we remove something from a namespace?

```
del WHAT
```

Deletes `WHAT` from the current namespace.

```
>>> del print
>>> print("asd")
asd
>>> del print
Traceback (most recent call last):
  File "<pyshell#198>", line 1, in <module>
    del print
NameError: name 'print' is not defined
>>>
```

The print we use is defined in another namespace (the builtin namespace). We can not mess with that.

Then there is the function namespace, which simply means each function has it's own namespace.

Note that *there is only one function namespace in the search list*, which is the namespace of the current function.

```
def testb():
    #function namespace of function testb
    print(a)

def testa():
    #function namespace of function testa
    a = 4
    testb()

testa()
```

If we run that program we get the following output.

```
========= OUTPUT =========
Traceback (most recent call last):
  File "/tmp/testprogram.py", line 10, in <module>
    testa()
  File "/tmp/testprogram.py", line 8, in testa
    testb()
  File "/tmp/testprogram.py", line 3, in testb
    print(a)
NameError: name 'a' is not defined
```

This is because in a was defined in the namespace of function `testa()` but when the execution is in the functions `testb()` the search uses:

1. builtin namespace
2. global namespace
3. function namespace of function `testb()`

We can use global variables in a function?

- Yes of course, as long as a variable with the same name does not overwrite it in the local namespace.
- If we really need to we can ''import" the global variable into a function namespace by using the `global` keyword.
  - This way we can change the variable
  - Very carefully think about if you can solve the problem in any other way. This use of global variables is horrendous to debug.

```
x = 1

def dontdothis():
    global x
    x = 20

dontdothis()
print(x)      # will output 20
```

What about getting information from function namespace to another function namespace?

- From the previous to the new: use function parameters.
- From the new to the previous: use the return statement.

Now what is *scope*?

- Merriam-Webster defines it as

  3 : extent of treatment, activity, or influence
  4 : range of operation: such as

- In a programming language similarly the scope of a variable is the range where it is defined and can be accessed.
  - In essence for python this coincides with the namespace.
  - A useful term when talking about where the variable can be accessed, i.e. what it's scope is.

Finally there is another type of namespace, which is a local namespace belonging to an object.

While not quite true you can imaging an object as a variable with an attached namespace,accessed by the `.` operator (as seen with the `import` statement).

We already used some objects, like string

Example:

```
>>> "asdf".upper()
'ASDF'
```

For the list of functions see the documentation.

Cleaning and Comparing Strings

A quick side note on comparing strings.

- Strings are case sensitive, for comparison if the content is more important than the case it is often useful to unify the case with either `.upper()` or `.lower()`.

  ```
  >>> mystring="Hallo"
  >>> mystring == "hallo"
  False
  >>> mystring.lower() == "hallo"
  True
  ```

- Space are part of a string. The function `.strip()` strips characters from the back and front of a string. By default it strips white spaces:

  ```
  >>> mystring_with_space = "hallo "
  >>> mystring_with_space == "hallo"
  False
  >>> mystring_with_space.strip() == "hallo"
  True
  ```

- Strip can be given a parameters string, each character in the string is stripped:

  ```
  >>> wohoo = "~*~* WOHOO *~*~"
  >>> wohoo.strip("~*")
  ' WOHOO '
  ```

  Notice however that the white space remain since we did not include them in the characters to strip!

- These commands can be combined:

  ```
  >>> wohoo.strip("~*").strip().lower()
  'wohoo'
  >>> wohoo.strip("~* ").upper()
  'WOHOO'
  ```

Functions and Parameters

So we just saw (`.strip()`) that functions can have default parameters. That sounds useful, how can we use that for our own functions?

The parameters given to a function are the variables we initialize in the function namespace. And just like we can assign values to variables we can assign values to variables in function namespaces (by using the `=` operator).

```
def greeter( name = "World" ):
    greetings = "Hello " + str(name)
    print(greetings)
```

Here we initialize name with `"World"`, so if we call:

```
>>> greeter()
Hello World
```

We can still change it as per usual:

```
>>> greeter("All!")
Hello All!
```

The regular version we used up until now simply means we want to have variable X but the caller of the function has to provide it's value. Both version can be combined of course.

```
def greeter2( name, prefix="Hello "):
    greeting = str(prefix)+str(name)
    print(greeting)
```

```
>>> greeter2()
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    greeter2()
TypeError: greeter2() missing 1 required positional argument: 'name'

>>> greeter2("World")
Hello World

>>> greeter2("All", "Dear ")
Dear All
```

Now consider this function, where we have two parameters with default values.

```
def greeter3( name="World", prefix="Hello "):
    greeting = str(prefix)+str(name)
    print(greeting)
```

What if we want to only change the `prefix`?

Well it has a name an we can just use that and write the usual kind of variable declaration.

```
>>> greeter3(prefix="Dear ")
Dear World
```

But once we use this declaration with names we can not go back to the ordered declaration without names!

```
>>> greeter3(prefix="Dear ", "All")
SyntaxError: positional argument follows keyword argument
```

Why is that? The interpreter does generally not know quite what to do so there is a rule:

1. Without names variables are filled from left to right.
2. Once names are used you have to continue using them.

So why is that? Consider the following function:

```
def example(A="A", B="B", C="C"):
    print(A + " " + B + " " + C)
```

Regular calls make sense, we just start left and then fill in the variables one by one.

```
>>> example("1")
1 B C
>>> example("1", "2")
1 2 C
>>> example("1", "2", "3")
1 2 3
```

So far so good, with names we are clear also:

```
>>> example(B="first")
A first C
```

But when we would say

```
example(B="first", "second")
```

Should the interpreter:

- Fill in A, starting from the beginning which would make sense.
- Fill in C, continuing from B which would also make sense.

The solution is the afore mentioned rule, we start from left to right as long as no variable names are involved. Then when specific variables are set we have to keep using variable names.

Please also note that the variable names do not have to be in order, this is perfectly fine:

```
>>> example(B="first", A="second")
second first C
```

Since this is basically a variable declaration for the function namespace we can of course also name variables without default values (just in case there was doubt):

```
>>> greeter2(name="and goodbye")
Hello and goodbye
```

Recall: `def greeter2(name, prefix = "Hello "):`

As a real example see the print function.

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

`print()` takes a number of objects and writes to to `sys.stdout` (the default output), one at a time.
In between each object is printed the `sep` string.
At the end the `end` string is printed (a new line per default).

```
>>> print("Hello", "World", sep='_sep_', end="\nThis is the END!!!!\n")
Hello_sep_World
```

```
This is the END!!!!
```

Functions and Return Values

Now we have the information about how to get information into a function.

But how about getting it out?

1. A function that returns nothing returns `None`.
   This means that there is no return statement.
2. The `return` statement ends the execution of the function and returns it's value.
   - A `return` statement without a value returns `None`
   - There can be multiple `return` statements!
   - The use of a variable to track the returned value vs. multiple `return` values is not obvious. But if in doubt choose the one which is easier readable.

Example: No return statement and empty return statements.

```
def no_return():
    pass

def empty_return():
    return

ret = no_return()
print("no_ret: " + str(ret) + " " + str(type(ret)))
ret = empty_return()
print("empty_ret: " + str(ret) + " " + str(type(ret)))
```

Gives:

```
no_ret: None <class 'NoneType'>
empty_ret: None <class 'NoneType'>
```

Example: Multiple returns.

```
def f(x,y):
    if y == 0:
        return
    return x/y

print( f(1,0) )
print( f(1,2) )
```

Gives:

```
None
0.5
```

This is often useful to have early returns for default values or error checks.

Assignment and Operator

When using while loops we have frequently encountered something like this:

```
N = N + 1
```

which gets tedious to write really fast.

There exists various forms of this type, meaning

> Do something to the variable and assign it back to the variable.

They are so frequent in fact that there is a special assignment operators to shorten these kind of expressions. So apart from the regular assignment operator `=` there also exists assignment operator of the type:

```
N += 1
```

Which is the same as `N = N + 1`.

The way to write this is the operator followed by `=` for assignment.

The following assignment operators exist:

Assignment	Regular	Mathematical
`N /= x`	`N = N / x`	$\frac{N}{x}$
`N %= x`	`N = N % x`	$N \mod x \equiv N - N\lfloor \frac{N}{x} \rfloor$
`N //= x`	`N = N // x`	$\lfloor \frac{N}{x} \rfloor$
`N -= x`	`N = N - x`	$N - x$
`N += x`	`N = N + x`	$N + x$
`N *= x`	`N = N * x`	$N \times x$
`N **= x`	`N = N ** x`	N^x

Formating Strings

Note: we will only discuss some basic uses of the string formatter here, there are lots of other type conversions and alignment options, see the documentation for this.

Strings also have a `.format()` function which allows formating a string. This is quite powerful, and often quite useful.

Basically it allows to replace a pair of brackets `{}` with a variable, the first `{}` is replaced with the first parameter to the format function, the second with the second, and so on.

```
>>> "This is a {}, and this is {}".format("test", "a second test")
'This is a test, and this is a second test'
```

Conversion is done implicitly:

```
>>> "This is a {}, and this is {}".format("test", 2)
'This is a test, and this is 2'
```

Compare the readability of the following two print statements:

```
>>> value = None
>>> print("Value: " + str(value) + " " + str(type(value)))
Value: None <class 'NoneType'>
>>> print("Value: {} {}".format( value, type(value) ))
Value: None <class 'NoneType'>
```

In case you want to repeat a specific variable you can also explicitly give the number of the variable in the brackets (**start counting at 0**).

```
>>> "Variable '{0}', variable '{1}', again variable '{0}'".format("Position 0", "Position 1")
"Variable 'Position 0', variable 'Position 1', again variable 'Position 0'"
```

If we want to use names instead of numbers we can:

```
>>> "Variable '{var1}', variable '{name2}', again variable '{var1}'".format(var1="Position 0",
 ↪  name2="Position 1")
"Variable 'Position 0', variable 'Position 1', again variable 'Position 0'"
```

Like with function parameters, named variables have to named and can not be used with numbers.

In case we want a literal `{}` we can put `{}` into the brackets like so `{{}}` this will print a literal `{}`.

Also take care, if you give too many parameters to format it does not matter, but too few are a problem:

```
>>> "Literal {{}} and format {}".format(1,2)
'Literal {} and format 1'
>>> "Literal {{}} and format {}, more {}, and too many {}".format(1,2)
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    "Literal {{}} and format {}, more {}, and too many {}".format(1,2)
IndexError: tuple index out of range
```

There is more we can do, for example explicit conversion. The format is `{VAR:FORMAT}`, where `VAR` is the number or name of a variable as before and `:FORMAT` gives the conversion specification. Most useful are `f` for float and `%` for percent, that is a float times 100.

```
>>> "float {0} in percent {0:%}".format(1/10)
'float 0.1 in percent 10.000000%'
```

The reason `f` is so useful that we can specify the precision of the output by giving the total number of digits and the number of digits after the decimal point. The format is `T.Af` where `T` is the total number of digits, `A` is the number of digits after the decimal point and `f` specifies that this is a float (also works for `%`).

```
>>> "float {0} in percent {0:%}".format(0.123456789)
'float 0.123456789 in percent 12.345679%'
>>> "float {0:5.3f} in percent {0:3.1%}".format(0.123456789)
'float 0.123 in percent 12.3%'
```

Also note that this conversion does round!

```
>>> "float {0:5.4f}".format(0.123456789)
'float 0.1235'
```

The total is the **minimum** number of characters used for the output of the float, including the decimal point.

Total is less than or equal to the number of characters required:

```
>>> print( "float {0:6.3f}\nfloat {0:7.3f}".format(123.45678) )
'float 123.457'
'float 123.457'
```

And higher:

```
>>> "float {0:10.3f}".format(123.45678)
'float    123.457'
```

Here we can see that the character allowance is filled up with white spaces. This allows for alignment of output. Here `d` is the integer format specifier and the decimal part is no needed.

```
>>> print("Should: {:6d}\nAnd Is: {:6d}".format(1234,567))
Should:   1234
And Is:    567
```

And specifically the decimal part of the float specifier allows to skip imprecisions in calculations:

```
>>> print("imprecise: {0}\nrounded  : {0:0.3f}".format(1/10*0.1))
imprecise: 0.010000000000000002
rounded  : 0.010
```

Note: The computer is based on a binary representation so the term $1/10$ has an infinite number of decimals, similar to $1/3 = 0.33333333...$ in the decimal system.

There is also a format string (referred to as *f-string*), basically a string and `.format()` combined. The string needs to be prefixed with an `f` (for format), then format the `{}` type output specification can directly use variables from accessible namespaces.

```
>>> anumber=1.2345
>>> print(f"This is the number: {anumber}")
This is the number: 1.2345
>>> print(f"This is the number: {anumber:4.2f}")
This is the number: 1.23
```

f-strings have a special shorthand for debugging, which prints the variable name as well as it's content. This is done by following the variable name in the format expression with a literal `=` .

```
>>> print(f"debug: {anumber=}")
debug: anumber=1.2345
```

Lists

Assume we want to write a function which can calculate the average temperature of a month based on the daily temperatures.

```
def average_temp_month(day1, day2, day3, day4, day5, day6, day7, day8, day9, day10, day11,
↪  day12, day13, day14, day15, day16, day17, day18, day19, day20, day21, day22, day23, day24,
↪  day25, day26, day27, day28, day29, day30, day31):
    #calculate
```

But what if it is February?

```
def average_temp_month(day1, day2, day3, day4, day5, day6, day7, day8, day9, day10, day11,
↪  day12, day13, day14, day15, day16, day17, day18, day19, day20, day21, day22, day23, day24,
↪  day25, day26, day27, day28, day29 = None, day30 = None, day31 = None):
    #calculate
```

And what if we want to calculate this on day 17 of the current month? We would need ifs for every day and so on.

Obviously there has to be a simpler way to handle such lists of values.

The answer: **Lists**

Lists are an *ordered* collection of values (the values do not need to be the same).

Values are separated by commas, started with `[` and ended with `]`.

```
>>> l = ["a", 1, 0.123, True] # create a list
>>> print(l)
['a', 1, 0.123, True]
```

The type is `list`.

```
>>> type(l)==list   # type
True
```

Basically everything we already know is true, `len()` is used to get the length of a list. A length 0 list is `False`, and if longer then 0 then it is `True`.

```
>>> len(l) # length of list
4
>>> bool([]) # empty list
False
>>> bool(l) # non empty list
True
```

Remember that a string is a list of character, so can we cast a string to a list?

```
>>> list("this is a string")
['t', 'h', 'i', 's', ' ', 'i', 's', ' ', 'a', ' ', 's', 't', 'r', 'i', 'n', 'g']
```

No we might want to access single items, or some of them.

This is done with an *index*, the index gives an *offset from the beginning of the list*. The offset is given as `variable[index]`.

So offset 0 is the first item:

```
>>> l[0]
'a'
>>> l[1]
1
```

This acts as a variable, so we can also change it:

```
>>> l
['a', 1, 0.123, True]
```

```
>>> l[1] = 2
>>> l
['a', 2, 0.123, True]
```

We can also go backward, and count from the end, negative indices are used for this.

```
>>> l[-1] # last item
True
>>> l[-2] # next to last item
0.123
```

Note however that in both cases we have to stay within the list:

```
>>> l[4]
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    l[4]
IndexError: list index out of range
>>> l[-5]
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    l[-5]
IndexError: list index out of range
```

We can also select a subset of the list, a **slice**, by giving two indices, separated by a color `:` like this `variabel[index1:index2]`. This will create a sublist starting with `index1` up to, but not including `index2`.

```
>>> l[1:3]
[2, 0.123]
```

If `index2` is omitted `len(variable)` is automatically used, generating a list from `index1` till the end.

```
>>> l[1:]
[2, 0.123, True]
```

If `index1` is omitted `0` is automatically used, generating a list from the beginning till `index2`.

```
>>> l[:3]
['a', 2, 0.123]
```

So far we know how to access and change items in a list, but we can also change the list.

`l.append(x)` adds `x` at the end of the list.

```
>>> l.append("this")
>>> l
['a', 1, 0.123, True, 'this']
```

`l.remove(x)` removes the first item in the list where the item is equal to `x`.

```
>>> l.remove(0.123)
>>> l
['a', 1, True, 'this']
```

`l.insert(i,x)` inserts `x` at position `i`.

```
>>> l.insert(2,0.123)
>>> l
['a', 1, 0.123, True, 'this']
```

`l.extend(x)` extends `l` with the contents of `x`.

```
>>> l.extend('2')
>>> l
['a', 1, 0.123, True, 'this', '2']
```

Note that this is different than `.append()` when it comes to list like types (python call these sequences or iterables).

```
>>> l=[1,2]
>>> g=[3,4]
>>> l.extend(g)
>>> l
[1, 2, 3, 4]
>>> l.append(g)
>>> l
[1, 2, 3, 4, [3, 4]]
```

Note that the 5th item in the list `l` is now a list!

```
>>> l[4]
[3, 4]
>>> l[4][1]
4
```

Lists can contain lists, and those lists can also contain lists and so on.

```
>>> l[4].append([5,6])
>>> l
[1, 2, 3, 4, [3, 4, [5, 6]]]
```

`l.clear()` deletes everything from the list.

```
>>> l.clear()
>>> l
[]
```

`l.reverse()` reverses the order of the items in the list.

```
>>> l =[1,2,7,4]
>>> l
[1, 2, 7, 4]
>>> l.reverse()
>>> l
[4, 7, 2, 1]
```

`l.pop(i=-1)` removes the i-th item in the list and also returns it.

```
>>> l.pop()
1
>>> l
[4, 7, 2]
```

`l.index(x, i=0, j=-1)` gives the index of the first item in the list which is equal to `x` and at or after index `i` but before index `j`.

```
>>> l.index(7)
1
>>> l.index(3)
Traceback (most recent call last):
  File "<pyshell#104>", line 1, in <module>
    l.index(3)
ValueError: 3 is not in list
```

`x in l` is `True` if at least one element in the list is equal to `x`.

```
>>> 7 in l
True
>>> 3 in l
False
```

`l.sort(key=None, reverse=False)` sorts the list by using the `<` operator. `key` can specify a function which is applied prior to sorting to each element of a list. If `reverse` is `True` `>` is used instead of `<`. This is the same as `.sort()` and then `.reverse()`.

```
>>> t = [1, "3", 4, 2]
>>> t
[1, '3', 4, 2]
>>> t.sort()
Traceback (most recent call last):
  File "<pyshell#111>", line 1, in <module>
    t.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
>>> t.sort(key=int)
>>> t
[1, 2, '3', 4]
>>> t.sort(key=str)
>>> t
[1, 2, '3', 4]
```

Note that the `key` function does not change the item, it is only used in conjunction with the sorting!

```
>>> l.sort(reverse=True)
>>> l
[7, 4, 2]
```

One more example with the `key` function.

```
>>> sl=list("Hello World")
>>> sl
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
>>> sl.sort()
>>> sl
[' ', 'H', 'W', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r']
>>> sl.sort(key=str.lower)
>>> sl
[' ', 'd', 'e', 'H', 'l', 'l', 'l', 'o', 'o', 'r', 'W']
```

`l.count(x)` Count the number of elements in `l` which are equal to `x`.

```
>>> l.append(7)
>>> l
[7, 4, 2, 7]
>>> l.count(7)
2
>>> l.count(4)
1
```

Remember: `l.count(x)` counts how many items in the list are equal to `x`! To get the total number of items in the list use `len(l)`.

Memory, Mutable, Immutable

`l.copy()` Return a shallow copy of the list.

```
>>> list1 = [1,2,3]
>>> listassign = list1
>>> listcopy = list1.copy()
>>> list1.append(4)
>>> print(list1)
[1, 2, 3, 4]
>>> print(listassign)
[1, 2, 3, 4]
>>> print(listcopy)
[1, 2, 3]
```

It is time we talked a bit about memory, mutable and immutable objects.

Objects are kept in memory and variables reference objects in memory. Basically a variable is just a human readable **reference** to a memory address.

The value that is represented by the variable in Python is stored at that memory address.

This is important because if two variables reference the same object in memory, the same memory address, and you change one of them, the other also changes. This was what is happening at the above example.

Some programming languages like Python abstract the memory management and handling to a large degree so we can mostly ignore it. We still have to be aware of it and be mindful of references as in the above example

This allows the programmer greater control over memory but comes at the cost of having to work with explicit memory management whether it is needed or not.

Two objects in Python can be equal, that is they have the same value, or they can be the same, as in they are really only one object which has two human readable names (variables) assigned to it.

In Python this can be tested with `is`.

```
>>> listcopy.append(4) # now all contain [1,2,3,4]
>>> list1 is listassign
True
```

```
>>> list1 is listcopy
False
>>> list1 == listassign
True
>>> list1 == listcopy
True
```

The `id(x)` function gives a unique integers for an object which is constant during it's lifetime. Specifically in the *CPython* implementation this gives the memory address of the object.

Now look at this

```
>>> a = 3
>>> b = a
>>> a is b
True
>>> b = 4
>>> a
3
>>> a is b
False
```

This is in essence the same as with the list example above, but with integers instead of lists.

The behaviour is different though.

The reason for this is the difference in mutable and immutable objects.

A **mutable** object can be changed, it will remain where it is in memory.

```
>>> l = [1,2,3]
>>> id(l)
139745426621192
>>> l.append(4)
>>> id(l)
139745426621192
```

An **immutable** object can not be changed. If it gets a new value a new object is generated (at a different address).

```
>>> a = 3
>>> id(a)
10914560
>>> a = 4
>>> id(a)
10914592
```

- Numbers are immutable:
  - int
  - float
  - bool
  - complex
- Strings are immutable.
- Bytes are a special type of integers (with values between 0 and 255) and are immutable.

Most other objects are mutable, but often there are immutable variants. If in doubt look at the documentation.

If you are unaware of this it can lead to errors.

```
def printitemize( plist):
    while plist:
        print(" - {}".format(plist.pop()))

l = ["Some", "Items"]
printitemize(l)
l.append("More")
print(l)
```

I we run this program we get:

```
======== OUTPUT ========
 - Items
 - Some
['More']
```

Remember that in `printitemize` we create a new variable `plist` and assign `l` to it `plist = l`! This is a mutable object and we changed it in the function!

```
def printitemizesave( plist):
    plist = plist.copy()
    while plist:
        print(" - {}".format(plist.pop()))

l = ["Some", "Items"]
printitemizesave(l)
l.append("More")
print(l)
```

Now we get what we expect.

```
======== OUTPUT ========
 - Items
 - Some
['Some', 'Items', 'More']
```

This is not the end of the story however!

`l.copy()` creates a new object with the same content of `l`, the content is assigned! Specifically if there are lists in it …

```
>>> list1=[ 1, 2, ["a","b"]]
>>> list2 = list1.copy()
>>> list2.append(3)
>>> list2
[1, 2, ['a', 'b'], 3]
>>> list1
[1, 2, ['a', 'b']]

>>> list2[2].append("c") # this is the list in the list!
>>> list2
```

```
[1, 2, ['a', 'b', 'c'], 3]
>>> list1
[1, 2, ['a', 'b', 'c']]
```

This is referred to as a **shallow copy**, i.e. only the topmost reference is copied.

In the modules `copy` there is a function `deepcopy` which creates a **deep copy**. That is a copy where the topmost and all included references are copied.

```
>>> list1=[ 1, 2, ["a","b"]]
>>> import copy
>>> list2 = copy.deepcopy(list1)
>>> list2[2].append("c")
>>> list2
[1, 2, ['a', 'b', 'c']]
>>> list1
[1, 2, ['a', 'b']]
```

Note (regarding lists): A slice creates a new list (shallow copy). So these are equivalent:

```
list2 = list1.copy()
list3 = list1[:] # remember the defaults of first and second index
```

Finally a note on **singletons**: Singletons are objects that can only every exist once!

An example is `None`, there is only one `None` object and whenever it is assigned to a variable that variable now points at that `None` object.

```
>>> def nothing():
    pass
>>> a = nothing()
>>> b = None
>>> a is b
True
```

There can be other singletons, depending in Python implementation, to preserve memory.

Low numbers typically are singletons!

```
>>> a=7
>>> b=7
>>> a is b
True
```

But they do not have to be

```
>>> a=12345
>>> b=12345
>>> a is b
False

>>> a = 1.1
>>> b = 1.1
>>> a is b
False
```

Only use `is` to compare mutable objects or `None`, never for immutable objects.

Tuples, Lists and Multi Assignments

There is an immutable version of list that is called a tuple. It uses round `()` instead of square `[]` brackets but can also be created implicitly by comma separation:

```
>>> t = ("a", 1, 0.123, True) # create a tuple
>>> print(t)
('a', 1, 0.123, True)
>>> t2 = "a", 1, 0.123, True # also creates a tuple
>>> print(t2)
('a', 1, 0.123, True)
```

The type is `tuple`.

```
>>> type(t) == tuple
True
```

As with lists, tuples of length zero are `False` and `True` otherwise, `len()` gives their length and they can easily be changed to lists and back.

```
>>> t
('a', 1, 0.123, True)
>>> l = list(t)
>>> l
['a', 1, 0.123, True]
>>> t2 = tuple(l)
>>> t2
('a', 1, 0.123, True)
```

They are immutable and can not be changed:

```
>>> t2.append(False)

Traceback (most recent call last):
  File "<pyshell#252>", line 1, in <module>
    t2.append(False)
AttributeError: 'tuple' object has no attribute 'append'
```

We can now also return multiple values from a functions by using list or tuples or implicit declaration of tuples.

```
def returnmany():
    return 1, 2, 1.23, True # implicit tuple

a = returnmany()
print(a)
```

Generates

```
========= OUTPUT =========
(1, 2, 1.23, True)
```

This is so useful that python even supports assigning multiple variables from a list, tuple or any other sequence!

By assigning a sequence to a list of variables we can assign the values in the list to the variables. Sounds more complicated than it is:

```
>>> t = ( 1, "a")
>>> a, b = t
>>> print("a = {} and b = {}".format(a,b))
a = 1 and b = a
```

The first variable `a` is assigned the first item in the tuple `1` and so on.

We can also use this with implicit tuple generation, like so

```
>>> a, b = b, a
>>> print("a = {} and b = {}".format(a,b))
a = a and b = 1
```

Where the right hand side `b,a` implicitly declares a tuple that is then assigned to `a` and `b`, in essence swapping the variables.

Be aware however that the numbers must match up.

```
>>> t = (1, 2, 3)
>>> a,b=t
Traceback (most recent call last):
  File "<pyshell#269>", line 1, in <module>
    a,b=t
ValueError: too many values to unpack (expected 2)
```

```
>>> t = (1, 2)
>>> a, b, c = t
Traceback (most recent call last):
  File "<pyshell#276>", line 1, in <module>
    a, b, c = t
ValueError: not enough values to unpack (expected 3, got 2)
```

Be aware of a special case when we want to declare a tuple with a single element.

```
>>> a = ( "tuple or not")
>>> a
'tuple or not'
```

In this case `a` is a string with contents “tuple or not”! This happens due to the multi assignment rule which applies the single element to the single variable.

If we want tuple with one element (the string “tuple or not”, we can add a trailing `,` like this

```
>>> b = ( "tuple or not",)
>>> b
('tuple or not',)
>>> len(b)
1
```

For Loops and Range

Now let us return to the function that generated this excursion, we want to calculate the average temperature in a month.

Well with lists this is simple.

```
def average_temp_month(temps = []):
    index = 0
    avg = 0
    while index < len(temps):
        avg += temps[index]
        index += 1
    if temps: # if len(temps) > 0:
        avg /= len(temps)
    else:
        avg = None
    return avg
```

This type of while loop is extremely frequent in programming, i.e. do something for each element of a list.

As per usual, when something is a tad longish to write and used extremely often it is simplified.

```
for VARIABLE in SEQUENCE:
    BLOCK
```

The BLOCK is executed once for each item in the sequence, which is assigned to the VARIABLE, starting from the beginning of the sequence to the end.

```
def average_temp_month(temps = []):
    avg = 0
    for day in temps:
        avg += day
    if temps:
        avg /= len(temps)
    else:
        avg = None
    return avg
```

Remember the `printitemize` function?

```
def printitemize( plist):
    for item in plist:
        print(" - {}".format(item))

l = ["Some", "Items"]
printitemize(l)
l.append("More")
print(l)
```

Results in

```
======== OUTPUT ========
 - Some
 - Items
['Some', 'Items', 'More']
```

Now remember this type of construct?

```
n = 1
while n < some_value:
    #do something
```

```
    n = n + 1
```

This is also used so often that can also be simplified to:

The tool for this is the built-in function `range(start=0, stop, step=1)` which generates a sequence of numbers starting from `start` until, but not including, `stop` with a step size of `step`.

Note: Range is neither a list nor a tuple it is a function which generates consecutive values on calls with the same parameters. This type of function is called an **iterable** function. It is a sequence however and can be converted into a list or tuple and be used in for loops. But this conversion has to happen explicitly.

Some examples:

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2,10))
[2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(2,10, 2))
[2, 4, 6, 8]
```

Negative steps are also possible is is the reversal of the sequence. Recall the default step size though!

```
>>> list(range(0,-10))
[]
>>> list(range(0,-10,-1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

So

```
n = 1
while n < some_value:
    #do something
    n = n + 1
```

can be rewritten as

```
for n in range(1, some_value):
    # do something
```

Note: Since the stop point is not included in the sequence `range( len(some_list))` generates indices for the first till last item of the list.

Some Built-in Functions for Sequences

So the average temperature requires us to sum up a list, a quite common task. And common tasks are simplified. There are actually a number of built-in functions that deal with sequences.

`sum(s)` Returns the sum of all elements of a sequence `s`.

```
def average_temp_month(temps = []):
    if not temps:
        return None
```

```
    return sum(temps)/len(temps)
```

```
>>> sum(range(10)) #numbers 0, 1, ... , 9
45
```

min(s, key=None) Returns the smallest value of the sequence/iterable s. A key function , similar to sort (see there), can be given with `key`.

max(s, key=None) Like min but returns the largest value

```
>>> min("Hello World")
' '
>>> max("Hello World")
'r'
>>> min(1,2,3,4,5,-3)
-3
>>> max(True, 1, "asdf")
Traceback (most recent call last):
  File "<pyshell#293>", line 1, in <module>
    max(True, 1, "asdf")
TypeError: '>' not supported between instances of 'str' and 'bool'
```

all(s) Returns `True` if all items in the sequence `s` are true. Returns `False` otherwise. This is like an `and` combination of all elements in the sequence.

any(s) Returns `True` if any of the items in the sequence `s` is true. Returns `False` otherwise. This is like an `or` combination of all the elements in the sequence.

```
>>> all("Hello World")
True
>>> any( range(10))
True
>>> all( range(10))
False
>>> all([[1], "length", True ])
True
```

filter(function, s) Create a new sequence by calling `function(item)` for each item in `s`. If the function returns `True` the item is included in the new sequence.

```
import random

def ispositive(number):
    if number >= 0:
        return True
    return False

listnum=[]
for i in range(10):
    listnum.append(random.randint(-20,20))
print(listnum)
print(list(filter(ispositive, listnum)))

========== OUTPUT ==========
[-2, 1, -19, -10, -2, -2, 16, 9, -19, 1]
[1, 16, 9, 1]
```

map(function, s) Create an iterable which contains result of the function applied to each item in s, basically map(func, [a, b, c, ..., z]) = [func(a),func(b),func(c), ... ,func(z)].

```
import random

def getsign(number):
    if number == 0:
        return "±"
    if number > 0:
        return "+"
    return "-"

listnum=[]
for i in range(10):
    listnum.append(random.randint(-20,20))
print("{}\n{}".format( listnum, list(map(getsign, listnum))))

========== OUTPUT ==========
[-5, 0, -1, -4, 17, 4, -14, -9, 15, -16]
['-', '±', '-', '-', '+', '+', '-', '-', '+', '-']
```

reversed(s) Return a reversed iterator over s, that is the last item of s is the first item of reversed(s).

```
>>> list( reversed( range(10)))
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

More on Strings

Strings are basic sequence types, we have already seen this. But strings are also a bit different, consequently they have a lot of functions specific to strings, see Documentation.

A few, which are frequently used in string handling, will be described here.

in This works slightly different for string in that it can search for strings also.

```
>>> "Hello" in "Hello World"
True
```

.find(substring, i=0, j=-1) This works like indexof for lists but can work with a string.

```
>>> "Hello World".find("Wo")
6
```

We have already seen that strings can be converted to lists like this:

```
>>> list("Hello World")
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
```

While this is nice it is not usually what is wanted. Normally we want to split a string at certain points. With the .split(separator) we can give a delimiter where to split a string. If we give space as the separator we can easily split the words in sentences.

```
>>> "Hello World".split(" ")
['Hello', 'World']
```

Of we could split based on , (see csv).

```
>>> "1,2,3,Hello World, some other value".split(",")
['1', '2', '3', 'Hello World', ' some other value']
```

The reverse of this is `STRING.join(somelist)` which joins a list to a string, adding the given `STRING` in between each pair of values.

```
>>> ";".join(['1', '2', '3', 'Hello World', ' some other value'])
'1;2;3;Hello World; some other value'

>>> "_".join(['Hello', 'World'])
'Hello_World'
```

Note that the value can also be an empty string.

```
>>> "".join(['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd'])
'Hello World'
```

Lambda Expressions / Anonymous Functions

We can sometimes give a function as a parameter, like with sort. This function is usually very specific and will not be reused. To simplify such cases anonymous functions can be used with the lambda keyword.

```
lambda PARAMETER_LIST: RETURN_VALUE
```

This syntax is functionally the same as

```
def ___NO NAME___(PARAMETER_LIST):
    return RETURN_VALUE
```

except that there really is no name.

Note that there is no block, all calculations have to be done in the expression which is returned.

Some examples, we have a list of 3-dimensional points (given as tuples):

```
>>> points
[(74, 70, 55), (61, 55, 61), (45, 15, 57), (84, 70, 38), (41, 38, 3)]
```

Normal sort will results in sorting by the first coordinate.

```
>>> points.sort()
>>> points
[(41, 38, 3), (45, 15, 57), (61, 55, 61), (74, 70, 55), (84, 70, 38)]
```

Assuming we want to sort by third coordinate we could do the following

```
>>> points.sort(key=lambda p:p[2])
>>> points
[(41, 38, 3), (84, 70, 38), (74, 70, 55), (45, 15, 57), (61, 55, 61)]
```

List Comprehension

List comprehension is simply a fast way to generate lists from other lists. The syntax is

```
[ LIST_GENERATION(ITEM) for ITEM in LIST ]
```

We iterate over `LIST`, setting `ITEM` to each element of list in order, just like a normal for loop. Then a new item is added to the generated list via `LIST_GENERATION` which is a normal statement and *may* use `ITEM`.

Two lists which are equal (`a==b` is `True`):

```
a = []
for ITEM in range(10):
    a.append( ITEM**2 )     # list_generation using ITEM

b = [ ITEM**2 for ITEM in range(10) ]
```

- The item does not have to be used in list generation of course, as an example let us generate a list of 100 random numbers (using the `random` module).

```
[random.random() for i in range(100)]
```

Note that the actual item `i` from the range is not used. It is just used as a counter.

- They can be nested.

Assume we want to generate a random list of three dimensional points.

```
[ [random.random() for j in range(3)] for i in range(100) ]
```

Here `[random.random() for j in range(3)]` is the `LIST_GENERATION` for the outer list and is itself a list comprehension style generation.

- There is a variant which uses conditionals, letting us allow a similar use that with filter. The syntax is

  ```
  [ LIST_GENERATION(ITEM) for ITEM in LIST if CONDITIONAL(ITEM) ]
  ```

The following constructs are equal, that is: `a==b` and `b==c` are `True`:

```
source = list(range(10))

a = [ i for i in source if i % 2 == 0 ]
b = list( filter( lambda i: i % 2 == 0, source) )
c = []
for i in source:
    if i % 2 == 0:
        c.append(i)

print(a == b)
print(b == c)
```

Dictionaries

Dictionaries in python are a mapping type, that is one value is associated with another. A dictonary is a collection of such mappings, **keys** and associated **value**.

Dictionaries can be defined in a similar way to lists but with `{` and `}` instead of `[` and `]`. Then a list of key values pairs, separated by comma can be given, a key value pair is separated by a collon `:`.

```
>>> d = {'key': 'value', "a number": 3, 3: "three"}
>>> print(d)
{'key': 'value', 'a number': 3, 3: 'three'}
```

The *value* can be everything a variable can be be, however the *key* must be immutable (string or number for example). The *key* can be of type tuple but only if everything in the tuple is immutable also.

The type is `dict`.

```
>>> type(d) == dict
True
```

A dictionary has a length, reflecting the number of key:value pairs it contains, and consequently it also has a truth value following the usual rules.

```
>>> len(d)
3
>>> bool({})
False
>>> bool(d)
True
```

Like with a list valuas in a dict can be read, changed and added. However instead of an index we use the key.

```
>>> d['key']
'value'
```

```
>>> d['key']=lambda i: i % 2 == 0
>>> d['key'](3)
False
```

```
>>> d['new']="added this to the dict"
>>> d
{'key': <function <lambda> at 0x7f61d5f8d158>, 'a number': 3, 3: 'three', 'new': 'added this to
↪  the dict'}
```

We can use `del` to delte a key (and associated value) from a dict.

```
>>> del d['key']
>>> d
{'a number': 3, 3: 'three', 'new': 'added this to the dict'}
```

Note that if a key i used it has to be in the dict, if not a `KeyError` exception is raised. We can check if a key is in the dict with `in` (and the reverse with `not in`).

```
>>> d
{'a number': 3, 3: 'three', 'new': 'added this to the dict'}
>>> 'key' in d
False
>>> 'key' not in d
```

```
True
```

- **`d.get(key, default=None)`** This function returns the key if it exists in the dict `d` or the default value (defaulting to `None`). This function therefor never raises an exception.

And of course a key can only exist once, so assigning to it again will overwrite the old value.

```
>>> d['new'] = 'overwritten'
>>> d
{'a number': 3, 3: 'three', 'new': 'overwritten'}
```

The default iterator of a dictionary is they list of keys, so conversion to list or looping is possible.

```
>>> list(d)
['a number', 3, 'new']
>>> for k in d:
    print("{}: {}".format(k, d[k]))

a number: 3
3: three
new: overwritten
```

`d.values()` Generates an iterator over the values in the dictionary.

```
>>> list(d.values())
[3, 'three', 'overwritten']
```

`d.items()` Generates an iterator over the items in the dictonary. An item is a tuple of the form `(key, value)`.

```
>>> list(d.items())
[('a number', 3), (3, 'three'), ('new', 'overwritten')]
```

`d.keys()` Generates the default iterator over the keys in the dict.

```
>>> list(d.keys())
['a number', 3, 'new']
>>> list(d)
['a number', 3, 'new']
```

Note that a dictonary is **not ordered** but has a lot of the same functionality of a list based on the **last in, first out** (LIFO) principle.

As such the list can not be sorted, but with the above iterators and the `sorted()` builtin function we can generate a sorted list of value or keys.

`sorted(iterable, key=None, reverse=False)` Return a new iterable which is a sorted version of the values in `iterable`. `key` can give a function to apply to each item befor sorting. If `reverse` is true the sorting is reversed.

```
>>> list(d.values())
[3, 'three', 'overwritten']

>>> [d[k] for k in sorted(d, key=lambda x: str(x).lower())]
['three', 3, 'overwritten']
```

Note the difference between `pop` and `popitem` as it is different from the way lists operate!

`d.pop(key[, default])` Returns the value of the key and removes the key from the dictorary. If the key does not exist it raises a KeyError exception except if default is set in which case it returns the default value.

`d.popitem()` Removes and returns a `(key, value)` tuple. **IMPORTANT**: Only from python 3.7 forward does this return based on LIFO, in older versions a random key-value pair is removed.

```
>>> d
{'a number': 3, 3: 'three', 'new': 'overwritten'}
>>> d.popitem()
('new', 'overwritten')
>>> d
{'a number': 3, 3: 'three'}
```

`d.clear()` Clears the dict, just like with lists.

`d.copy()` Copies the dict, just like list (this is a *shallow copy*).

`d.update( some_dict)` Like `extend()` for list. Basically this adds all items from `some_dict` to `d`, overwriting existing keys.

```
>>> d
{'a number': 3, 3: 'three'}
>>> e={3:2,4:4}
>>> d.update(e)
>>> d
{'a number': 3, 3: 2, 4: 4}
```

Dictionaries, List Assignment and For Loops

A quick note: List assignments works in for loops.

```
>>> d.items()
dict_items([('a number', 3), (3, 2), (4, 4)])
>>> list(d.items())
[('a number', 3), (3, 2), (4, 4)]
>>> for key, val in d.items():
    print("{} -> {}".format(key,val))

a number -> 3
3 -> 2
4 -> 4
```

The same conditions as with regular list assignment holds, i.e., number of variables and list items must be equal. Recall that this is the same as:

```
items = d.items()
i = 0
while i < len(items):
    key, val = items[i]
    print("{} -> {}".format(key,val))
    i += 1
```

Dictionary Comprehension

Dictionaries can be used in a similar fashion to how lists can be used for list comprehension.

Only two changes need to be kept in mind:

- `[` and `]` for lists are replaced with `{` and `}` for dictionaries
- Elements in the dictionary need to be key-value pairs given as `key:value`

```
>>> a = { "{}^2".format(i):i**2 for i in range(5) }
>>> a
{'0^2': 0, '1^2': 1, '2^2': 4, '3^2': 9, '4^2': 16}
>>> b = { k:v for k,v in a.items() if v % 2 == 0 }
>>> b
{'0^2': 0, '2^2': 4, '4^2': 16}
```

Modules Include in Python

There are a large number of modules included in python, we can not cover all of this here.

For the full overview see the documentation for The Python Standard Library.

This is just a brief list of modules, which are especially useful, related to things we covered in the course or the assignments.

copy *Shallow and deep copy operations*

math *Mathematical functions*
: Lots of math functions, from asin to sqrt, constants like pi and so on.

random *Generate pseudo-random numbers*
: Also allows to shuffle lists or get random elements from lists.

statistics *Mathematical statistics functions*
: Mean, median, standard variation, all the standard stuff.

itertools *Functions creating iterators for efficient looping*
: More fancy iterators fro looping like looping over all permutation of lists, of joining two lists and iterating at the same time.

operator *Standard operators as functions*
: These are very useful for filter and map like operations, but can also help with lambda functions.

datetime *Basic date and time types*
: If you want to measure time, or know what date it is.

os *Miscellaneous operating system interfaces*
: Allows for typical interface stuff with an os like change directories, create or delete them, get user or group id and so on.

os.path *Common pathname manipulations*
: Specifically allows parts of path to be joined together with either `/` or `\` depending on whether the system is linux or windwos in an automatic fashion. Otherwise allows to check if files exist and so on.

Reading and Writing Files

The built in function `open()` is used to to interact with files. It is commonly given two arguments: `open(filename, mode)`. `filename` is the path to a file (string) and `mode` gives the type of file and whether we read from or write to it.

The filename is straightforward, but to be OS agnostic the use of the `os` and `os.path` modules is quite useful.

The mode on the other hand is a bit more complicated, it consists of two parts.

1. What is the type of the file (in essence do we read strings from a text file or bytes from a binary file).

mode — file type	meaning
t	text mode (**default**)
b	binary file mode

2. How do we interact with a file, read or write. Do we want to add to the file or overwrite it etc.

   The `+` can be added to `rwax`.
   Only one type and one interaction can be given, e.g., `"r+t"`, `"w"` (which defaults to `"wt"`) or `"ab"`.

mode — read / write	meaning
r	open for reading (**default**)
w	open for writing, empties the file!
a	open for writing, append at the end of the file
x	open and create a file, fails if file exists
.+	opens for reading *and* writing

`open()` returns a *file like* object. This only means that the returned object has certain functions implemented. For text file (`t`) these are described here.

The most important part is:

- `read(size=-1)` which reads `size` characters and returns them as a string. If size is negative or None the whole file is read.
- `readline(size=-1)` which reads a line, or a maximum of size characters. A line is until a newline is encountered or the file ends.
- `write(s)` writes the string `s` to the file and returns the number of characters actually written.
- `close()`closes the file.

A file like object also behaves as a sequence, giving each line as an entry.

As an example let us assume we write the python program `readself.py`:

```
flo = open("readself.py") #filelikeobject
for line in flo:
    print(line)
```

```
flo.close()
```

The file will open it's own source, read it one line after another and print it. Note that the print appends a newline and the newline from the original file is kept, so newlines will double.

Note also that there are implementation in the standard library which deal with common types of files, like

- `csv` for reading and writing comma separated values (CSV) files.
- `configparser` for reading configuration files.

A (Superbrief) Introduction to Classes

A class is in essence an object (with an object namespace!) that is used to group functions and data.

```
class NAME:
    BLOCK
```

Classes can be used like that, however the idea is that a class is a *template* for this object. And an actual object of that type can be initialized by using all the class structure and function but initializing it with specific data.

For example a class might describe a train, store the number of seats in this train, the origin and destination etc. However, in reality there are multiple trains,with different number of seats, hailing from different cities and traveling to, again, different cities.

A class `Train` might describe the archetypical train and a variable `trainA` might be of that class (as a type) and specifically travel from Salzburg to Vienna with a capacity for 50 people.

So assume we have such a `trainA` which is an object generated from the `Train` class. If we store the number of available `seats` in the class, how do we access that?

- `trainA` is an object and has an object namespace
- To access an object namespace we need the object reference (and then add the `.` operator).
- We have to pass the object to every function in the class so we can access it.

The syntax for this is a bit weird since python abstracts half of it.

- Python automatically inserts a reference to the **instance object** as first parameter to each function call if the function belongs to the object! (This does not hold true for the **class**, see later).
- However, the function has to explicitly state the self reference (we need that to access the namespace after all). The convention is to call this `self`.

Assume we want to write a function `getSeats()` which returns the number of seats for a train.

The function definition would look something like this, we explicitly have to state the self reference as first variable.

```
class Train:
    def getSeats(self):
        return self.seats
```

But the call to a `trainA` object of this class would look like this:

```
trainA.getSeats()
```

And python would automatically insert `trainA` as the first parameter of the function call such that self then references this object.

Now then, how do we generate an **instance object** from a **class** with or without initial values. The syntax is to call the class like a function, this does a couple of things:

1. It generates a new object.
2. Everything in the class namespace is referenced (careful, this is a reference not a copy). Specifically don't declare variables in the namespace of the class, use `__init__` (see next point) to generate the variable in the object.
3. A special function to initialize this new object is called `__init__()`. Parameters passed to the generataing class will be passed to the `__init__` function and python will inject a self reference at the beginning.

So

```
trainA = Train()
```

generates an object `trainA` from the class `Train`.

Let us assume that we want to give a train an origin, destination and number of seats (by using an initializer).

We need to write a function `__init__` which takes a self reference and the parameters.

```
class Train:
    def __init__(self, origin, destination, seats):
        self.orig = origin
        self.dest = destination
        self.seats = seats
```

No we can properly generate a train, that goes from a city to a city and has seats

```
trainA = Train("Salzburg", "Vienna", 50) # the self reference is added by python automatically.
```

Note: The initializer function is often called *constructor* since it constructs the new object from the class.

What if we want to print information about our train?

```
>>> trainA = Train("Salzburg", "Vienna", 50)
>>> print(trainA)
<__main__.Train object at 0x7f037db8a2e8>
```

Well, we could implement a function `print` which prints the class.

However there is a better way: Python does duck typing, so we just have to tell our class to quack like a string!

Python of course does not know how to do this, so we have to take care of this as programmers. There are interfaces for a lot of conversions which are tried by python to see if it can convert an object to a given type.

Specifically for string conversion python tries to call a function `__str__()` which is supposed to return a string version of the object.

```
class Train:
    def __str__(self):
        return "Train from {} to {} with {} seats".format(self.origin, self.destination,
        ↪  self.seats)
```

Now if we try this again python can call `__str__` and convert the class to a string for printing.

```
>>> trainA = Train("Salzburg", "Vienna", 50)
>>> print(trainA)
Train from Salzburg to Vienna with 50 seats
```

What actually happens is that `str(object)` is converted to `object.__str__()`. That allows the use of standard Python operators to convert our class instead of having to know the name we gave to the conversion function.

```
>>> s = str(trainA)
>>> print(s)
Train from Salzburg to Vienna with 50 seats
```

The class `Train`, just so that it is completely present.

```
class  Train:
    def __init__(self, origin, destination, seats):
        self.orig = origin
        self.dest = destination
        self.seats = seats

    def getSeats(self):
        return self.seats

    def __str__(self):
        return "Train from {} to {} with {} seats".format(
                    self.orig,
                    self.dest,
                    self.seats
                    )
```

Now a remark on the difference between **instance object** and **class** on the basis of the following:

```
class A:
    l=[1,2]

    def p(self):
        for e in self.l:
            print(e)


def b(self):
    while self.l:
        print(self.l.pop())
```

Let us instantiate two **instance objects** B and C from **class** A.

```
B=A()
C=A()
```

So far so normal, but `B.l` and `C.l` are reference to `A.l`!

```
B.l.append(3) #  B.l -> reference to A.l
print("=== C.p() ===")
C.p() # C.l -> reference to A.l
```

outputs

```
=== C.p() ===
1
2
3
```

Solution: Create object specific variables in the `__init__` function!

Functions are references too!

```
A.p = b #also only a reference
print("=== B.p() ===")
B.p() # call to b, will empty B.l which is A.l
print("=== C.p() ===")
C.p() # Now C.l whihc is A.l is empty
```

Generates output:

```
=== B.p() ===
3
2
1
=== C.p() ===
```

Solution: None really, just don't do that. It makes code hard to read!

And finally only **instance objects** auto insert the self reference.

```
>>> A.p()
Traceback (most recent call last):
  File "/tmp/classref.py", line 25, in <module>
    A.p()
TypeError: b() missing 1 required positional argument: 'self'
```

But

```
B.p()
```

works fine, except it does not print anything since A.l is empty.

This is only a brief overview, what is mainly missing is this:

- Inheritance, which allows classes to use the template from another class and build on it.
- Interface functions, similar to `__str()__` which allows classes to behave like sequences, iterators, lists or dictionaries.

For more on classes start with a look at the Class Documentation.

For more on the data models and emulations of types see Special Named Methods Documentation.