Python programming for beginners

Stefan Zhauryd
Instructor

Module 8

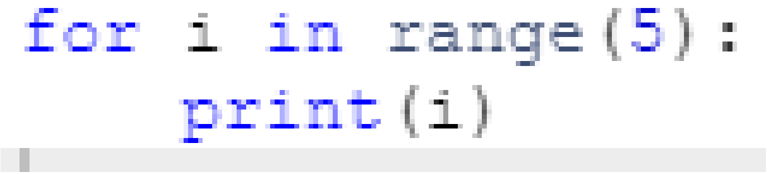
Miscellaneous

In this module, you will learn about:

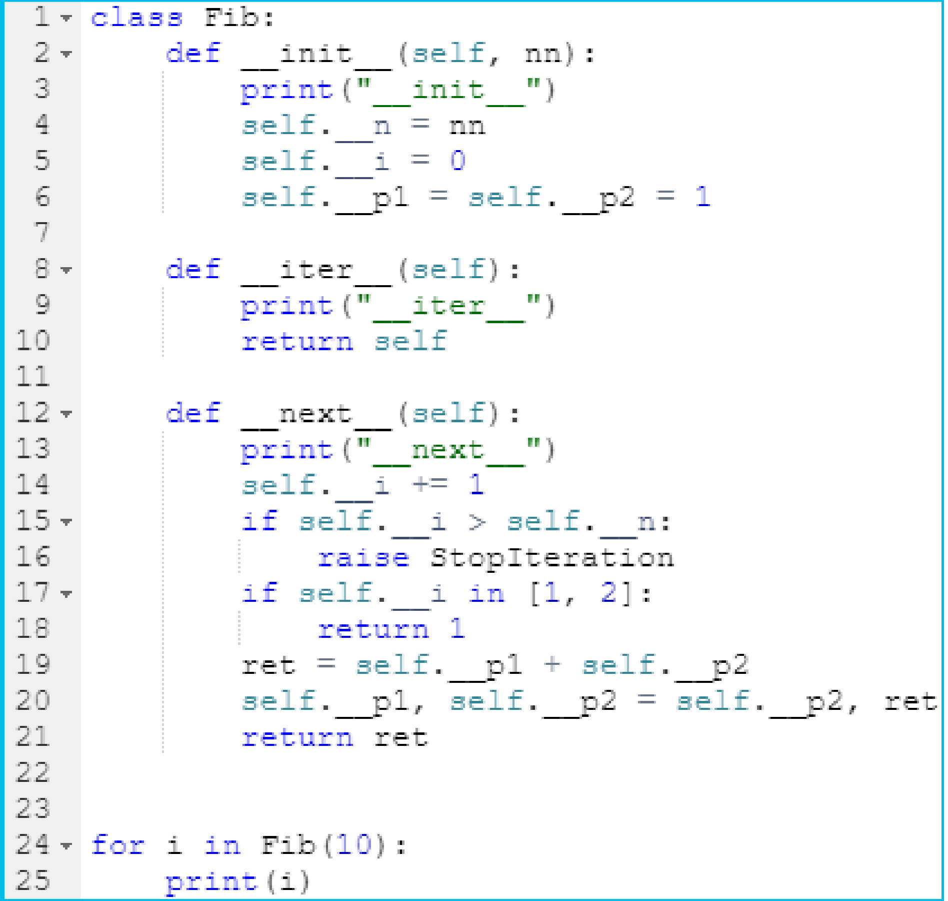
- Generators, iterators and closures;
- Working with file-system, directory tree and files.

Generators - where to find them

A Python generator is a piece of specialized code able to produce a series of values, and to control the iteration process.



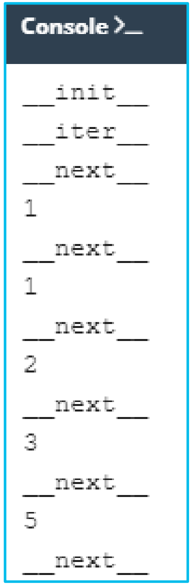
```
for i in range(5):
    print(i)
```



```python
class Fib:
    def __init__(self, nn):
        print("__init__")
        self.__n = nn
        self.__i = 0
        self.__p1 = self.__p2 = 1

    def __iter__(self):
        print("__iter__")
        return self

    def __next__(self):
        print("__next__")
        self.__i += 1
        if self.__i > self.__n:
            raise StopIteration
        if self.__i in [1, 2]:
            return 1
        ret = self.__p1 + self.__p2
        self.__p1, self.__p2 = self.__p2, ret
        return ret


for i in Fib(10):
    print(i)
```

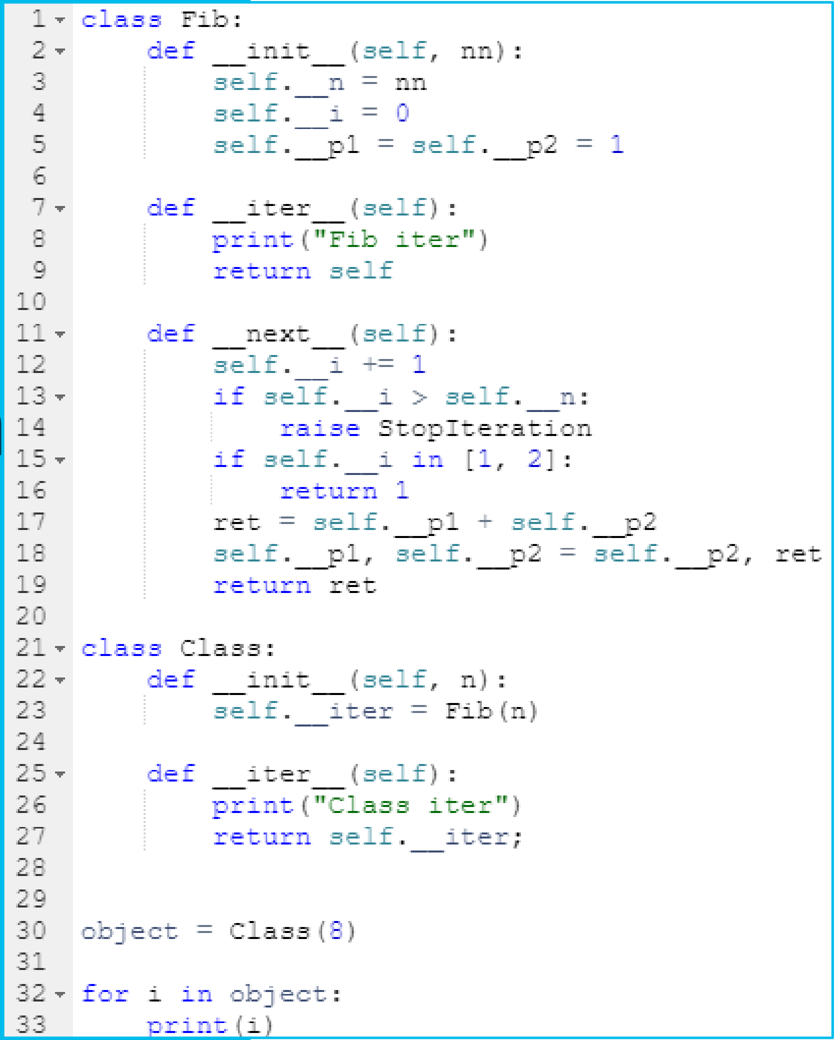
An iterator must provide two methods:

- **__iter__()** which should return the object itself and which is invoked once (it's needed for Python to successfully start the iteration)
- **__next__()** which is intended to return the next value (first, second, and so on) of the desired series - it will be invoked by the for/in statements in order to pass through the next iteration; if there are no more values to provide, the method should raise the **StopIteration exception**.

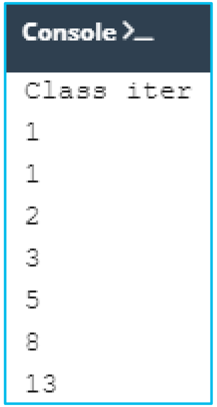


```
__init__
__iter__
__next__
1
__next__
1
__next__
2
__next__
3
__next__
5
__next__
```

Generators - wh to find them: continued

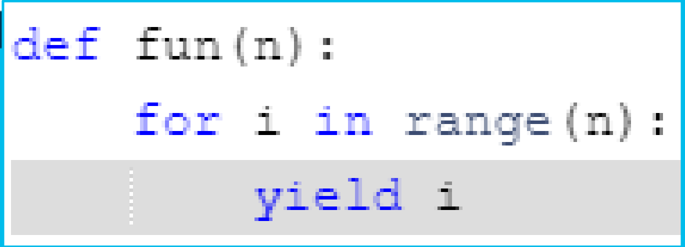


```
class Fib:
    def __init__(self, nn):
        self.__n = nn
        self.__i = 0
        self.__p1 = self.__p2 = 1

    def __iter__(self):
        print("Fib iter")
        return self

    def __next__(self):
        self.__i += 1
        if self.__i > self.__n:
            raise StopIteration
        if self.__i in [1, 2]:
            return 1
        ret = self.__p1 + self.__p2
        self.__p1, self.__p2 = self.__p2, ret
        return ret

class Class:
    def __init__(self, n):
        self.__iter = Fib(n)

    def __iter__(self):
        print("Class iter")
        return self.__iter;


object = Class(8)

for i in object:
    print(i)
```

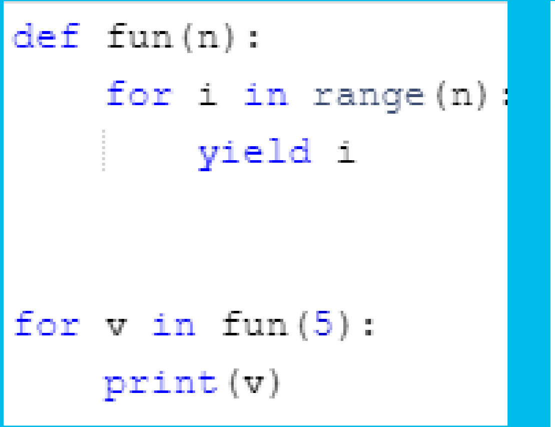


```
Class iter
1
1
2
3
5
8
13
```

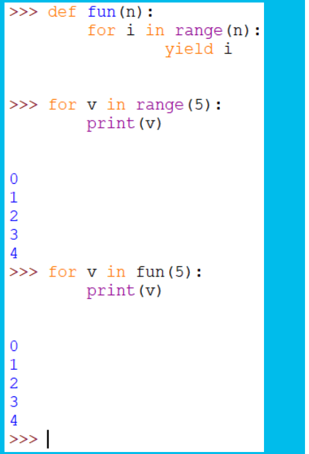
The **yield** statement



```
def fun(n):
    for i in range(n):
        yield i
```



```
def fun(n):
    for i in range(n):
        yield i



for v in fun(5):
    print(v)
```



```
>>> def fun(n):
        for i in range(n):
                yield i


>>> for v in range(5):
        print(v)



0
1
2
3
4
>>> for v in fun(5):
        print(v)



0
1
2
3
4
>>> |
```

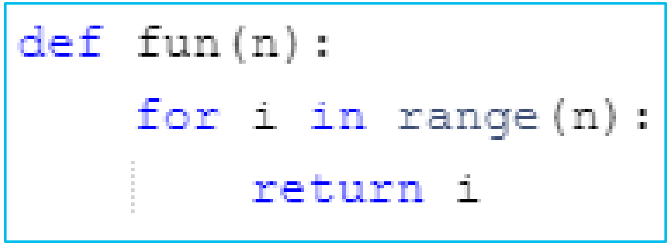
I've added yield instead of return. This little amendment turns the function into a generator, and executing the yield statement has some very interesting effects.

First of all, it provides the value of the expression specified after the yield keyword, just like return, but doesn't lose the state of the function.

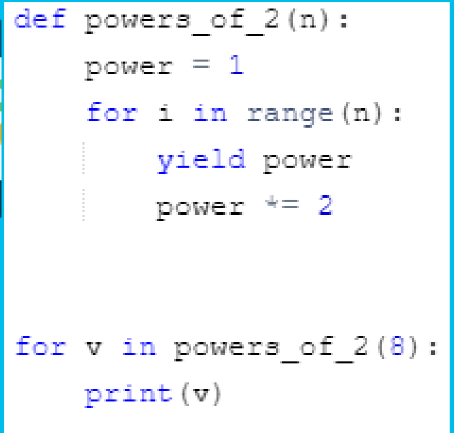
All the variables' values are frozen, and wait for the next invocation, when the execution is resumed (not taken from scratch, like after return).

There is one important limitation: such a function should not be invoked explicitly as - in fact - it isn't a function anymore; it's a generator object.

The invocation will return the object's identifier, not the series we expect from the generator.

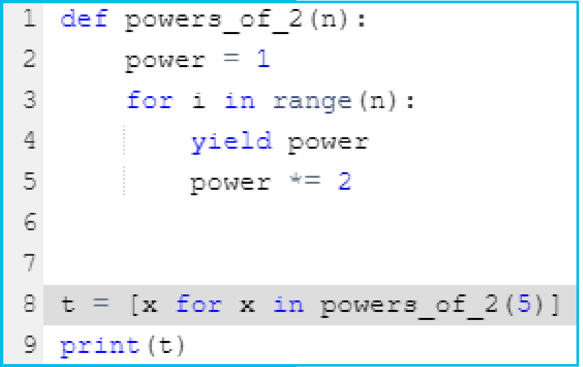


```
def fun(n):
    for i in range(n):
        return i
```

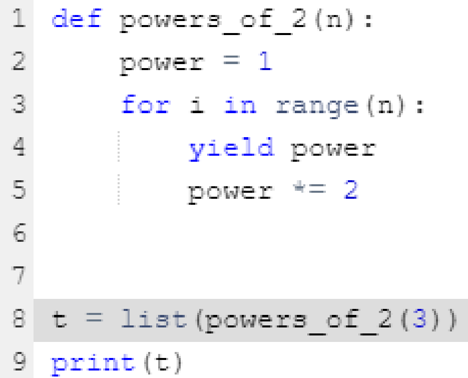


```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2


for v in powers_of_2(8):
    print(v)
```

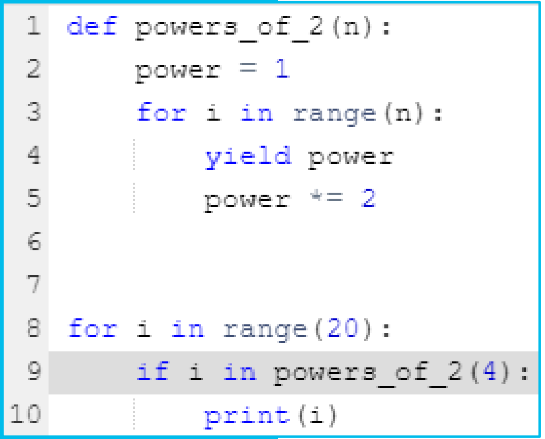
How to build your own generator



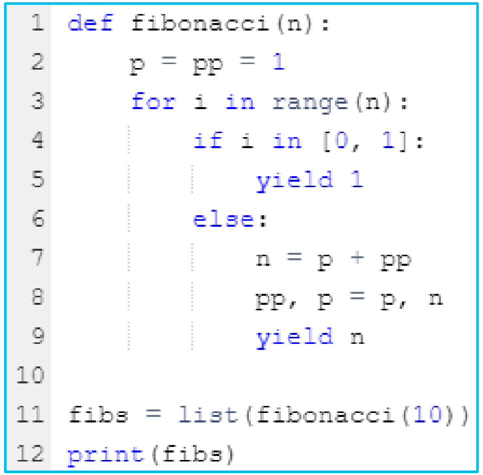
```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2


t = [x for x in powers_of_2(5)]
print(t)
```



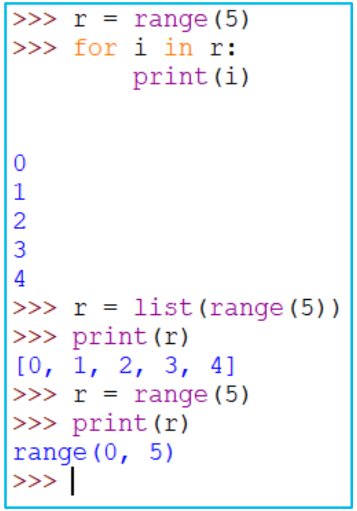
```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2


t = list(powers_of_2(3))
print(t)
```



```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2


for i in range(20):
    if i in powers_of_2(4):
        print(i)
```

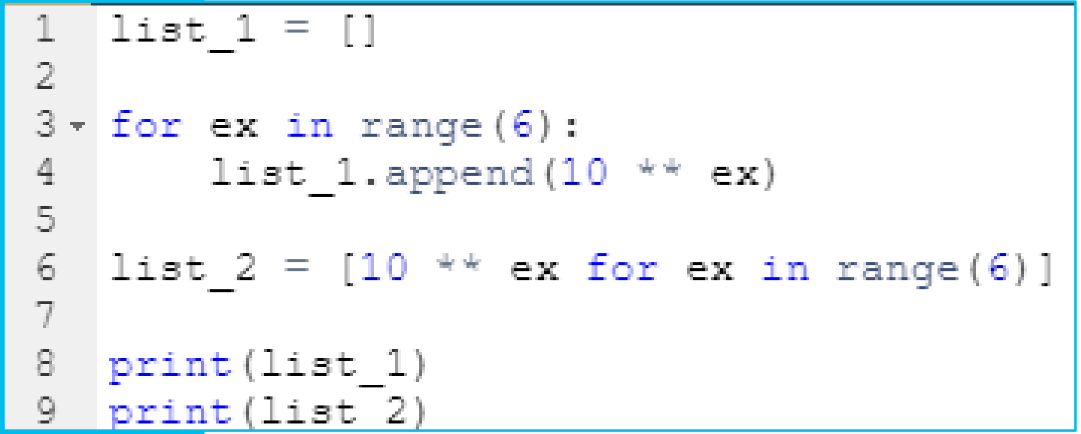


```
def fibonacci(n):
    p = pp = 1
    for i in range(n):
        if i in [0, 1]:
            yield 1
        else:
            n = p + pp
            pp, p = p, n
            yield n

fibs = list(fibonacci(10))
print(fibs)
```

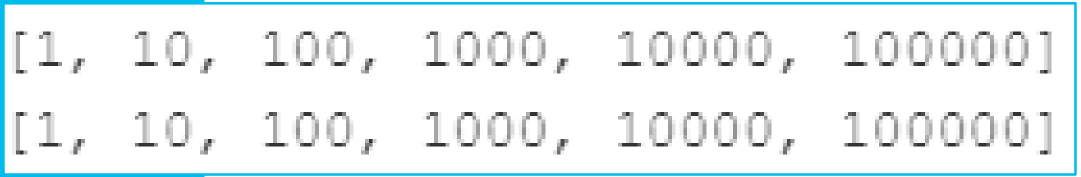


```
>>> r = range(5)
>>> for i in r:
        print(i)



0
1
2
3
4
>>> r = list(range(5))
>>> print(r)
[0, 1, 2, 3, 4]
>>> r = range(5)
>>> print(r)
range(0, 5)
>>> 
```

More about list comprehensions

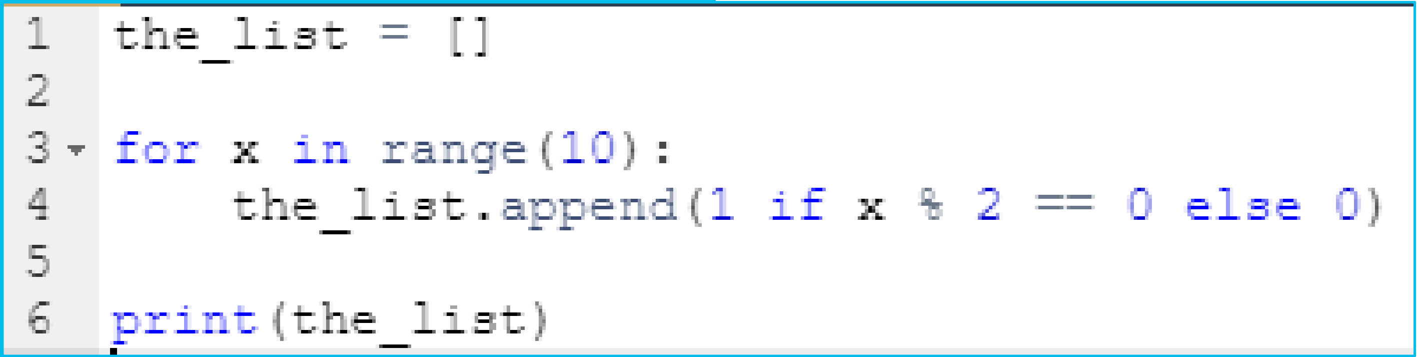


```python
list_1 = []

for ex in range(6):
    list_1.append(10 ** ex)

list_2 = [10 ** ex for ex in range(6)]

print(list_1)
print(list_2)
```

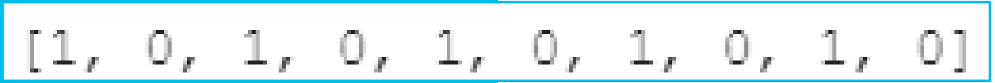


```
[1, 10, 100, 1000, 10000, 100000]
[1, 10, 100, 1000, 10000, 100000]
```

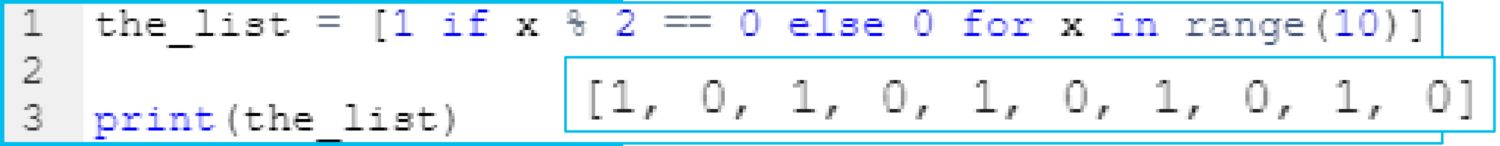
More about list comprehensions: continued



```python
the_list = []

for x in range(10):
    the_list.append(1 if x % 2 == 0 else 0)

print(the_list)
```

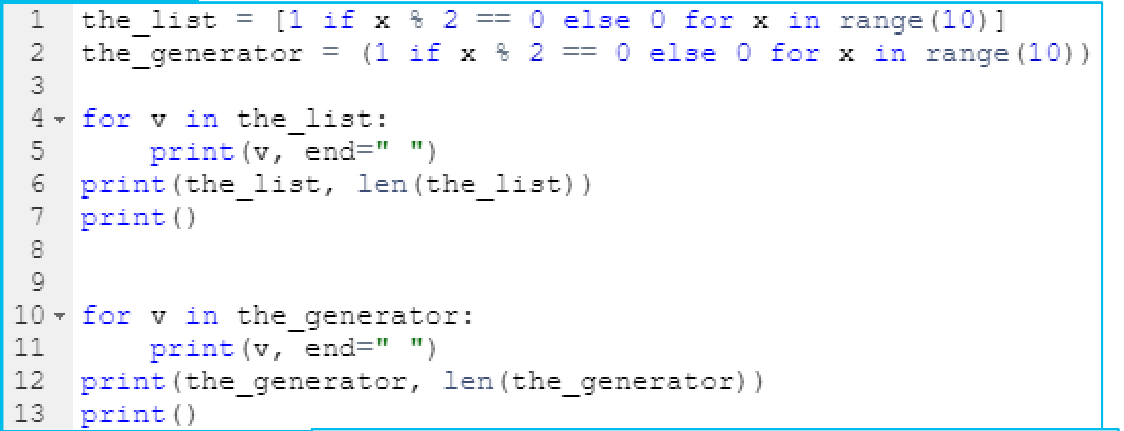


```
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
```



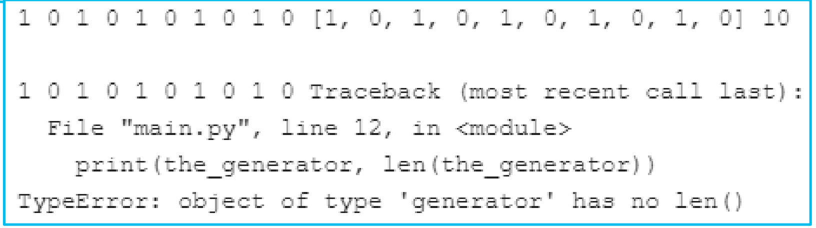
```python
the_list = [1 if x % 2 == 0 else 0 for x in range(10)]

print(the_list)
```

```
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
```

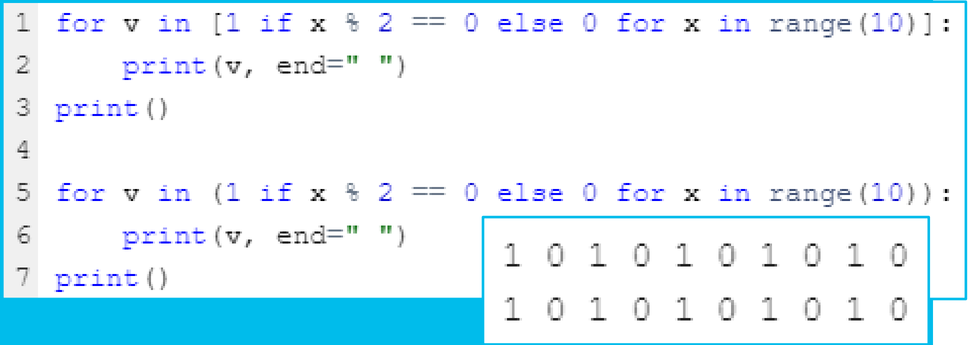


```python
the_list = [1 if x % 2 == 0 else 0 for x in range(10)]
the_generator = (1 if x % 2 == 0 else 0 for x in range(10))

for v in the_list:
    print(v, end=" ")
print(the_list, len(the_list))
print()


for v in the_generator:
    print(v, end=" ")
print(the_generator, len(the_generator))
print()
```

More about list comprehensions: continued



```
1 0 1 0 1 0 1 0 1 0 [1, 0, 1, 0, 1, 0, 1, 0, 1, 0] 10

1 0 1 0 1 0 1 0 1 0 Traceback (most recent call last):
  File "main.py", line 12, in <module>
    print(the_generator, len(the_generator))
TypeError: object of type 'generator' has no len()
```



```python
for v in [1 if x % 2 == 0 else 0 for x in range(10)]:
    print(v, end=" ")
print()

for v in (1 if x % 2 == 0 else 0 for x in range(10)):
    print(v, end=" ")
print()
```

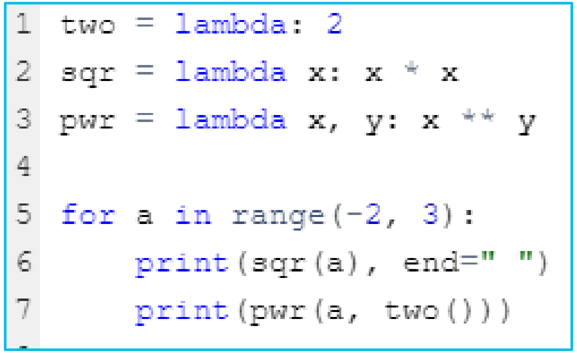
```
1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0
```

The **lambda** function

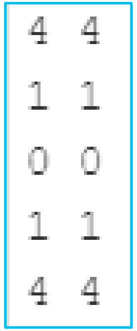
A **lambda function** is a function without a name (you can also call it an anonymous function).

lambda *parameters*: *expression*

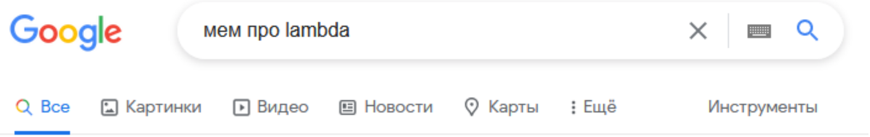
Such a clause returns the value of the expression when taking into account the current value of the current lambda argument.



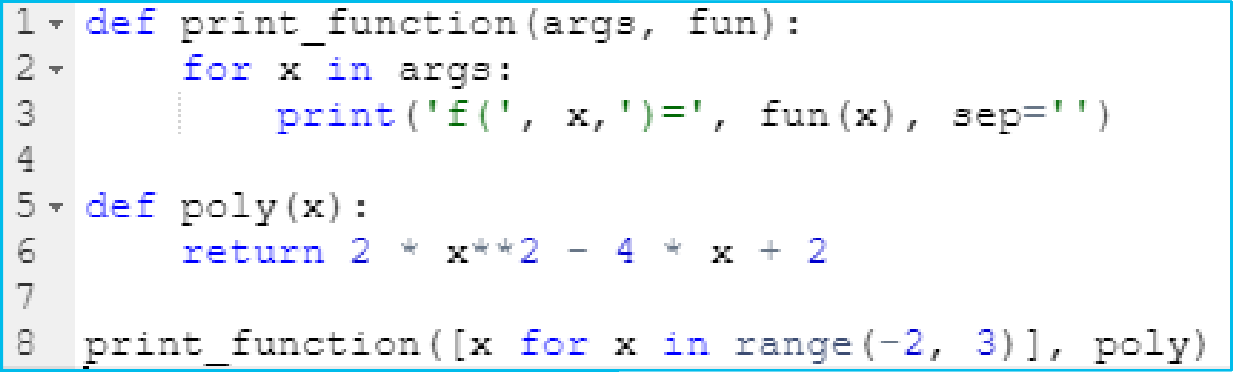
```
two = lambda: 2
sqr = lambda x: x * x
pwr = lambda x, y: x ** y

for a in range(-2, 3):
    print(sqr(a), end=" ")
    print(pwr(a, two()))
```



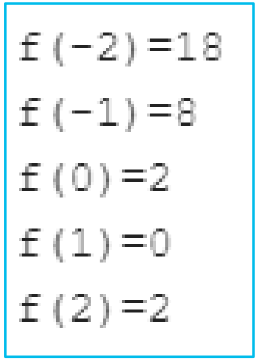
```
4 4
1 1
0 0
1 1
4 4
```



мем про lambda

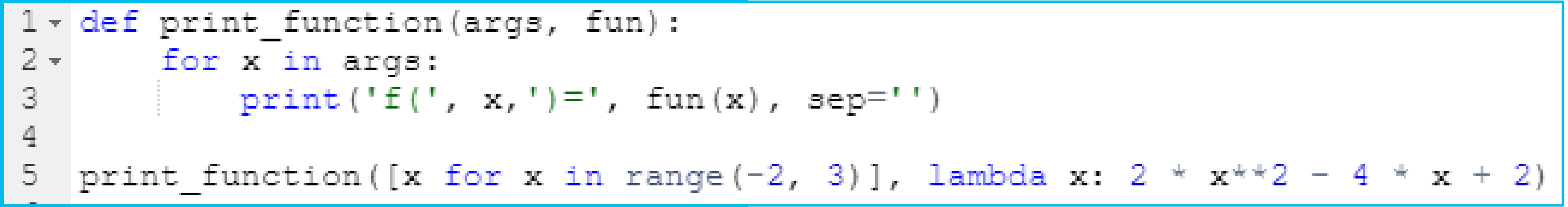


```python
def print_function(args, fun):
    for x in args:
        print('f(', x,')=', fun(x), sep='')

def poly(x):
    return 2 * x**2 - 4 * x + 2

print_function([x for x in range(-2, 3)], poly)
```



```
f(-2)=18
f(-1)=8
f(0)=2
f(1)=0
f(2)=2
```

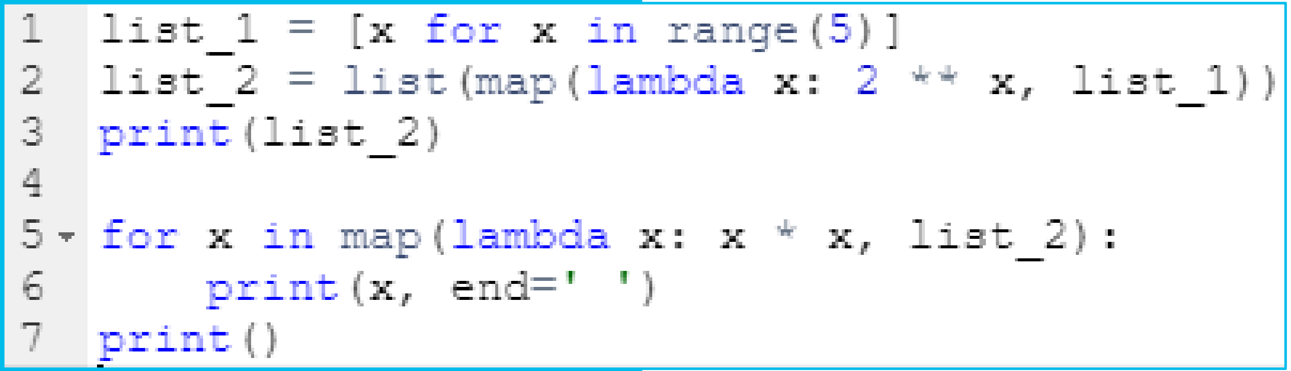
How to use lambdas and what for?



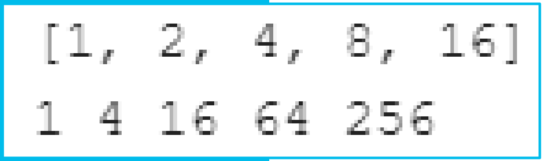
```python
def print_function(args, fun):
    for x in args:
        print('f(', x,')=', fun(x), sep='')

print_function([x for x in range(-2, 3)], lambda x: 2 * x**2 - 4 * x + 2)
```

Lambdas and the **map() function**

The **map() function** applies the function passed by its first argument to all its second argument's elements, and **returns an iterator delivering all subsequent function results**.



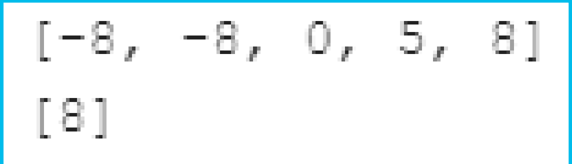
```
list_1 = [x for x in range(5)]
list_2 = list(map(lambda x: 2 ** x, list_1))
print(list_2)

for x in map(lambda x: x * x, list_2):
    print(x, end=' ')
print()
```



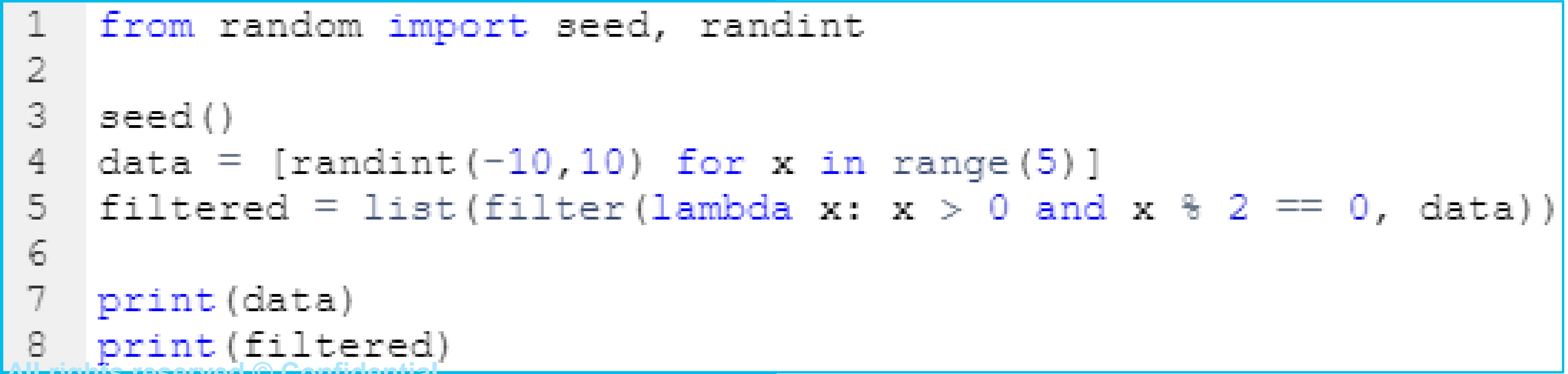
```
[1, 2, 4, 8, 16]
1 4 16 64 256
```

Lambdas and the **filter() function**

It expects the same kind of arguments as map(), *but does something different* - **it filters its second argument while being guided by directions flowing from the function specified as the first argument (the function is invoked for each list element, just like in map()).**



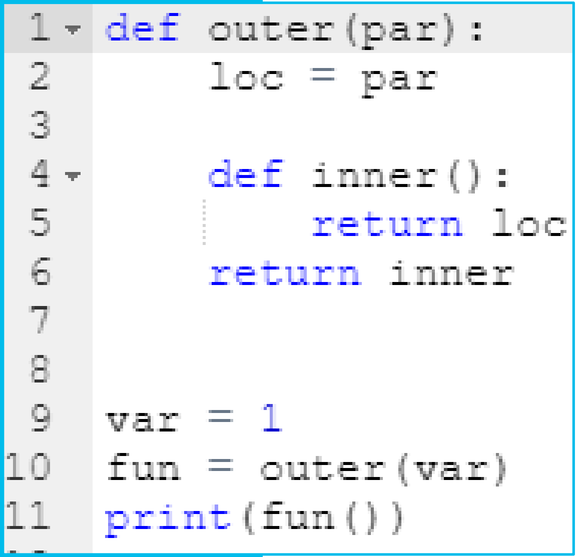
```
[-8, -8, 0, 5, 8]
[8]
```



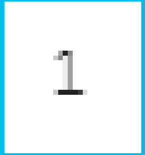
```
from random import seed, randint

seed()
data = [randint(-10,10) for x in range(5)]
filtered = list(filter(lambda x: x > 0 and x % 2 == 0, data))

print(data)
print(filtered)
```

A brief look at **closures**

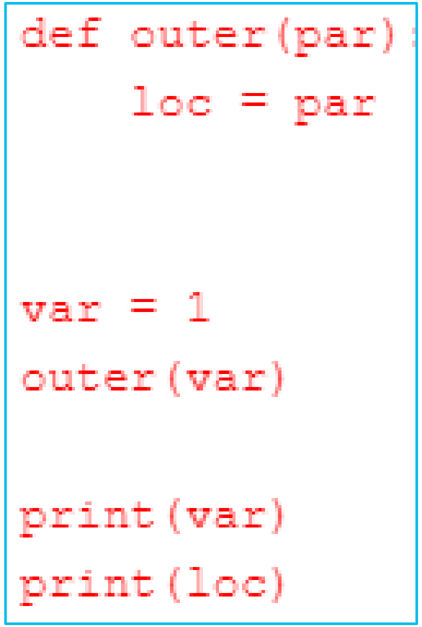
The **closure** is a technique which allows the storing of values in spite of the fact that the context in which they have been created does not exist anymore.



```python
def outer(par):
    loc = par

    def inner():
        return loc
    return inner


var = 1
fun = outer(var)
print(fun())
```

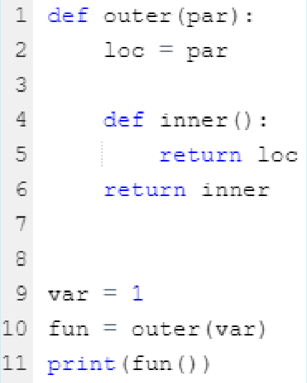


```
1
```

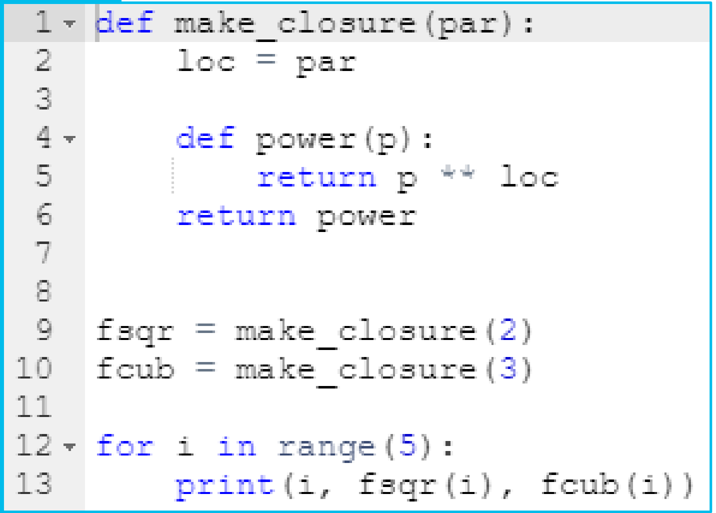


```python
def outer(par):
    loc = par



var = 1
outer(var)


print(var)
print(loc)
```

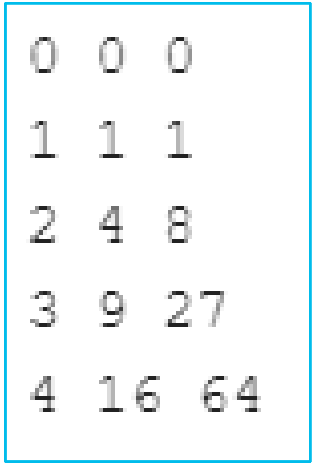
A brief look at closures: continued



```python
def outer(par):
    loc = par

    def inner():
        return loc
    return inner


var = 1
fun = outer(var)
print(fun())
```



```python
def make_closure(par):
    loc = par

    def power(p):
        return p ** loc
    return power


fsqr = make_closure(2)
fcub = make_closure(3)

for i in range(5):
    print(i, fsqr(i), fcub(i))
```



```
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
```

Key takeaways

1. An iterator is an object of a class providing at least two methods (not counting the constructor!):

- **__iter__()** is invoked once when the iterator is created and returns the iterator's object itself;
- **__next__()** is invoked to provide the next iteration's value and raises the StopIteration exception when the iteration comes to and end.

2. The **yield statement** can be used only inside functions. The yield statement suspends function execution and causes the function to return the yield's argument as a result. Such a function cannot be invoked in a regular way – its only purpose is to be used as a generator (i.e. in a context that requires a series of values, like a for loop.)

Key takeaways

3. A conditional expression is an expression built using the if-else operator. For example:

```
print(True if 0 >=0 else False)#outputs True.
```

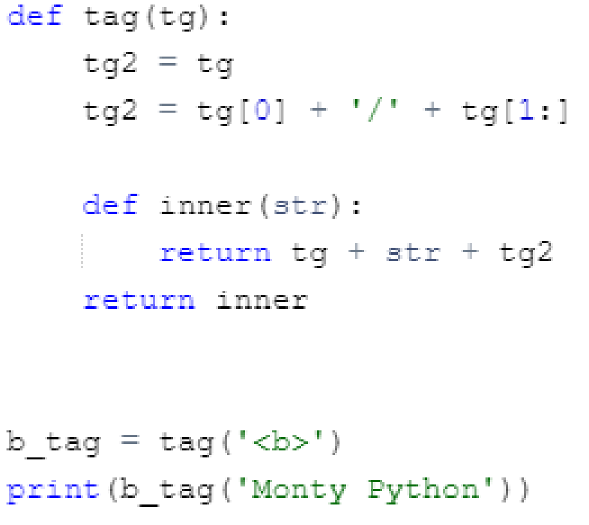
4. A list comprehension becomes a generator when used inside parentheses (used inside brackets, it produces a regular list). For example:

```
for x in (el * 2 for el in range(5)):
    print(x) #outputs 02468.
```

5. **The map(fun, list) function** creates a copy of a list argument, and applies the fun function to all of its elements, returning a generator that provides the new list content element by element. For example:

```
short_list = ['mython', 'python', 'fell', 'on', 'the', 'floor']

new_list = list(map(lambda s: s.title(), short_list))

print(new_list)#['Mython', 'Python', 'Fell', 'On', 'The', 'Floor'].
```

Key takeaways



```
def tag(tg):
    tg2 = tg
    tg2 = tg[0] + '/' + tg[1:]

    def inner(str):
        return tg + str + tg2
    return inner


b_tag = tag('<b>')
print(b_tag('Monty Python'))
```

6. The **filter(fun, list) function** creates a copy of those list elements, which cause the fun function to return True. The function's result is a generator providing the new list content element by element. For example:

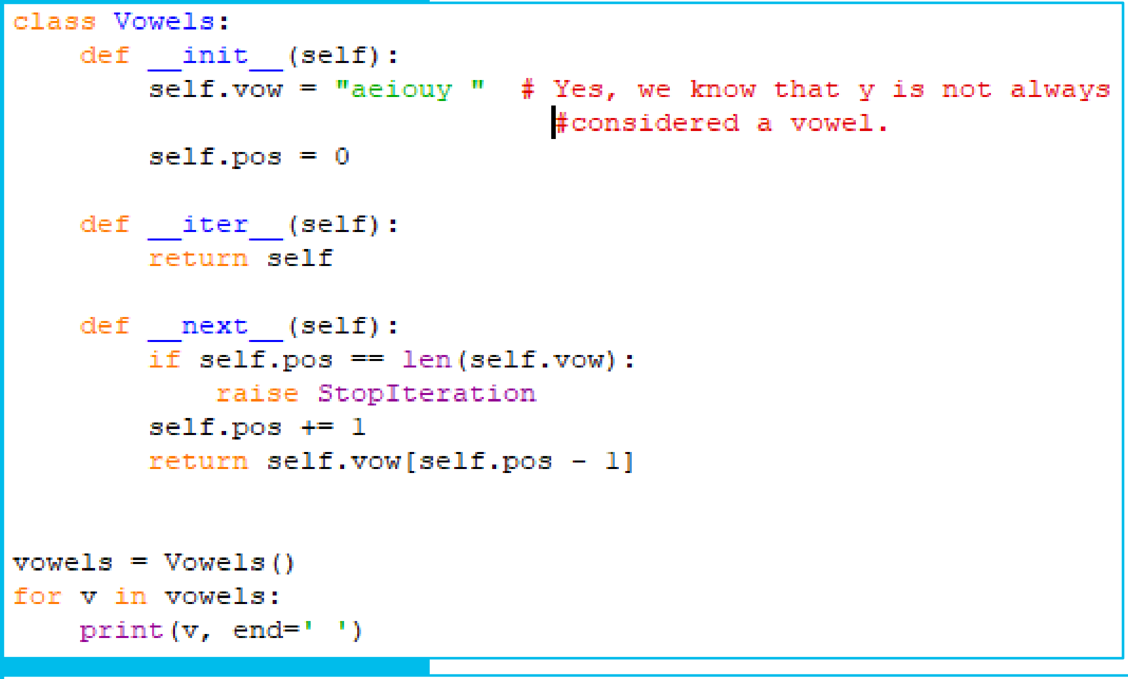
short_list = [1, "Python", -1, "Monty"]

new_list = list(filter(lambda s: isinstance(s, str), short_list))

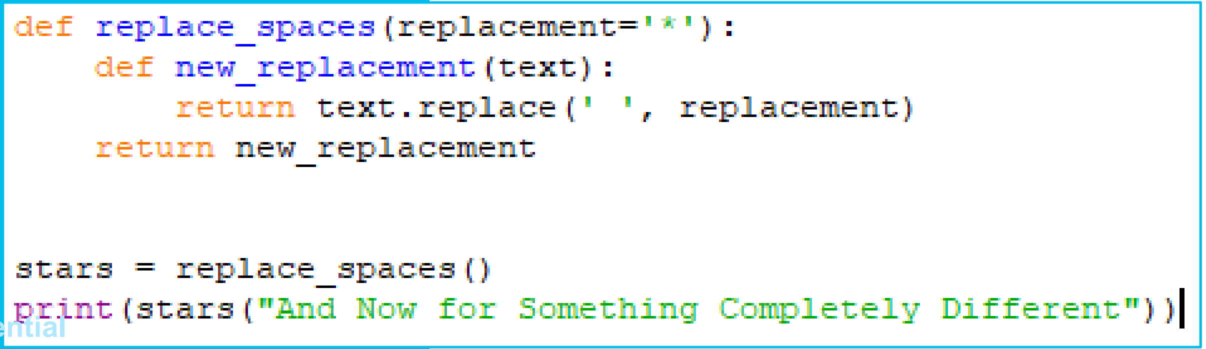
print(new_list)*#outputs ['Python', 'Monty'].*

7. A **closure** is a technique which allows the storing of values in spite of the fact that the context in which they have been created does not exist anymore. For example:

Examples



```
class Vowels:
    def __init__(self):
        self.vow = "aeiouy "  # Yes, we know that y is not always
                                #considered a vowel.
        self.pos = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.pos == len(self.vow):
            raise StopIteration
        self.pos += 1
        return self.vow[self.pos - 1]


vowels = Vowels()
for v in vowels:
    print(v, end=' ')
```



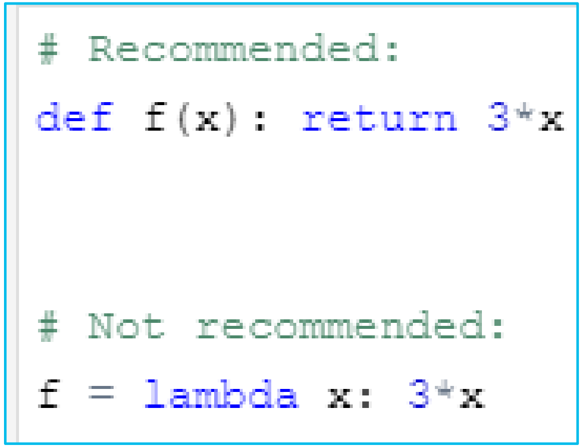
```
def replace_spaces(replacement='*'):
    def new_replacement(text):
        return text.replace(' ', replacement)
    return new_replacement


stars = replace_spaces()
print(stars("And Now for Something Completely Different"))
```

Note

The Style Guide for Python Code, recommends that lambdas should not be assigned to variables, but rather they should be defined as functions.

https://www.python.org/dev/peps/pep-0008/

This means that it is better to use a def statement, and avoid using an assignment statement that binds a lambda expression to an identifier. For example:

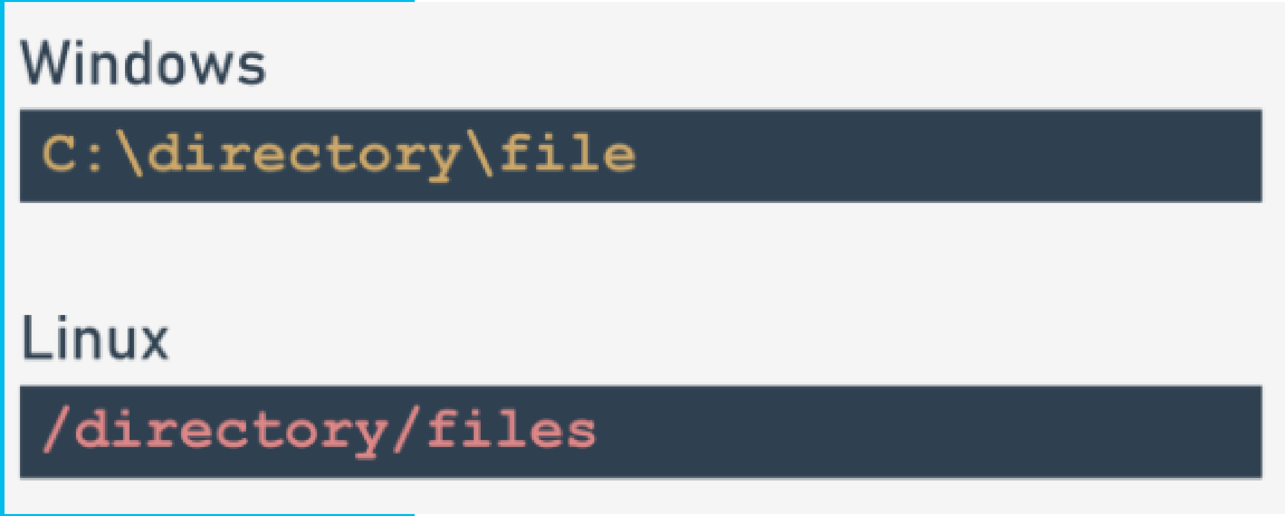


```
# Recommended:
def f(x): return 3*x


# Not recommended:
f = lambda x: 3*x
```

Accessing files from Python code

All rights reserved © Confidential

File names



Windows

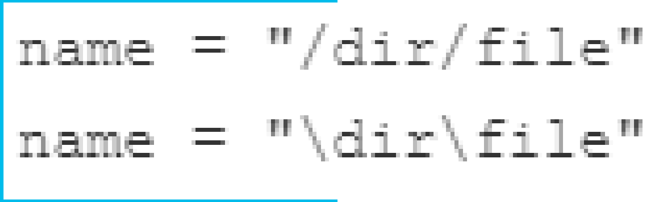
```
C:\directory\file
```

Linux

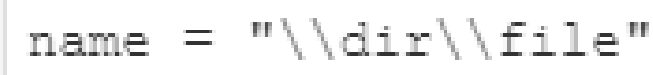
```
/directory/files
```

File names: continued

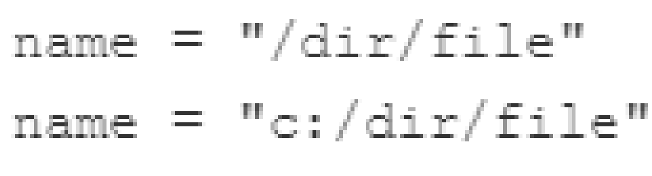
The operation of connecting the stream with a file is called **opening the file**, while disconnecting this link is named **closing the file**.



```
name = "/dir/file"

name = "\dir\file"
```

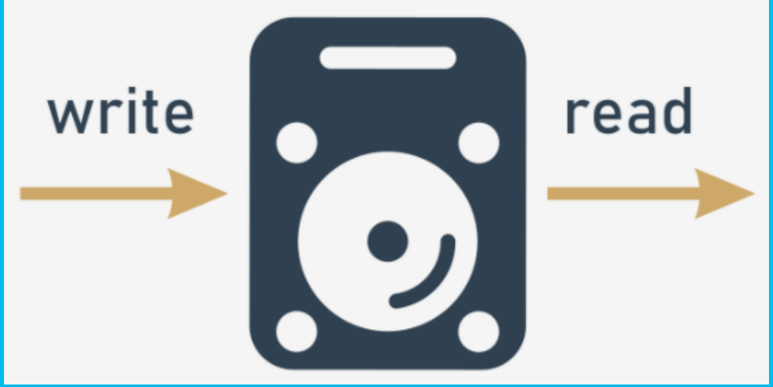


```
name = "\\dir\\file"
```



```
name = "/dir/file"

name = "c:/dir/file"
```

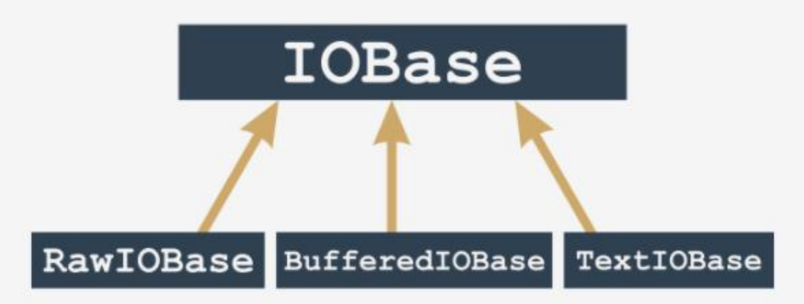
File streams



There are three basic modes used to open the stream:

- **read mode:** a stream opened in this mode allows read operations only; trying to write to the stream will cause an exception (the exception is named UnsupportedOperation, which inherits OSError and ValueError, and comes from the **io** module);
- **write mode:** a stream opened in this mode allows write operations only; attempting to read the stream will cause the exception mentioned above;
- **update mode:** a stream opened in this mode allows both writes and reads.

File handles



You never use constructors to bring these objects to life. The only way you obtain them is to invoke the function named **open()**

The function analyses the arguments you've provided, and automatically creates the required object.

If you want to get rid of the object, you invoke the method named **close()**

The invocation will sever the connection to the object, and the file and will remove the object.

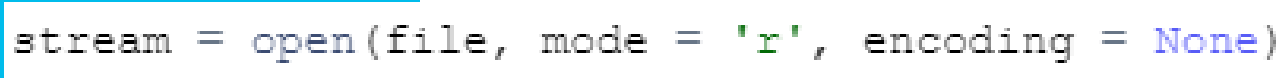
File handles: continued portable?

In Unix/Linux systems, the line ends are marked by a single character named LF (ASCII code 10) designated in Python programs as \n.

Other operating systems, especially these derived from the prehistoric CP/M system (which applies to Windows family systems, too) use a different convention: the end of line is marked by a pair of characters, CR and LF (ASCII codes 13 and 10) which can be encoded as \r\n.

File handles: continued \n

The opening of the stream is performed by a function which can be invoked in the following way:



```
stream = open(file, mode = 'r', encoding = None)
```

Opening the streams: modes

r open mode: **read**

- the stream will be opened in read mode;
- the file associated with the stream must exist and has to be readable, otherwise the open() function raises an exception.

w open mode: **write**

- the stream will be opened in write mode;
- the file associated with the stream doesn't need to exist; if it doesn't exist it will be created; if it exists, it will be truncated to the length of zero (erased); if the creation isn't possible (e.g., due to system permissions) the open() function raises an exception.

a open mode: **append**

- the stream will be opened in append mode;
- the file associated with the stream doesn't need to exist; if it doesn't exist, it will be created; if it exists the virtual recording head will be set at the end of the file (the previous content of the file remains untouched.)

r+ open mode: **read and update**

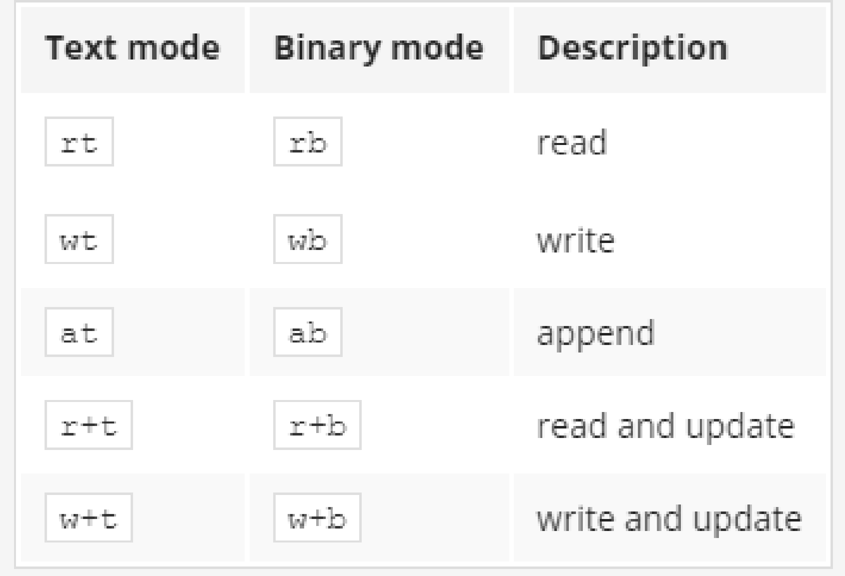
- the stream will be opened in read and update mode;
- the file associated with the stream must exist and has to be writeable, otherwise the open() function raises an exception;
- both read and write operations are allowed for the stream.

w+ open mode: **write and update**

- the stream will be opened in write and update mode;
- the file associated with the stream doesn't need to exist; if it doesn't exist, it will be created; the previous content of the file remains untouched;
- both read and write operations are allowed for the stream.

Selecting text and binary modes

You can also open a file for its exclusive creation. You can do this using the **x open mode**. If the file already exists, the **open()** function will raise an exception.



Text mode	Binary mode	Description
`rt`	`rb`	read
`wt`	`wb`	write
`at`	`ab`	append
`r+t`	`r+b`	read and update
`w+t`	`w+b`	write and update

Opening the stream for the first time

Pre-opened streams

import sys

sys.stdin

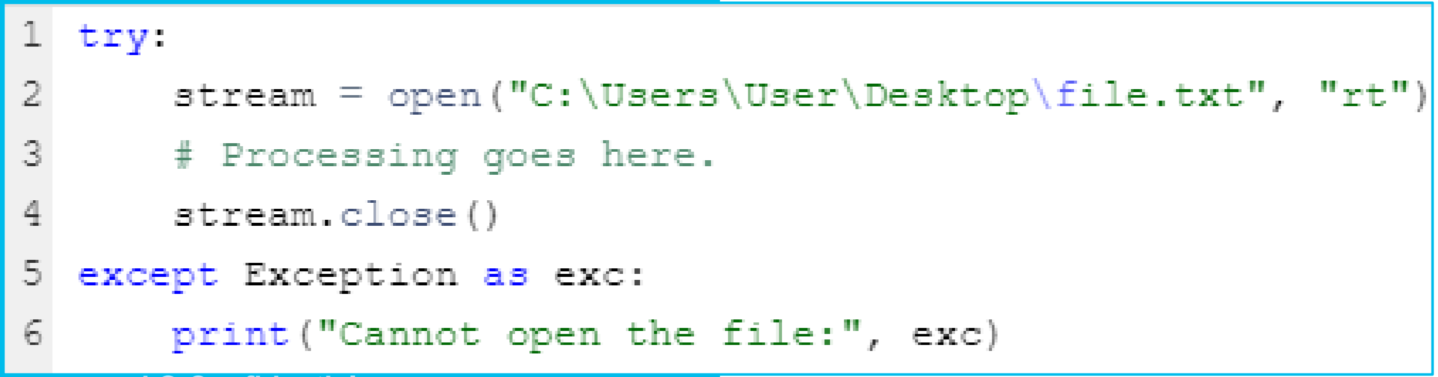
stdin (as standard input)

sys.stdout

stdout (as standard output)

sys.stderr

stderr (as standard error output)



```
try:
    stream = open("C:\Users\User\Desktop\file.txt", "rt")
    # Processing goes here.
    stream.close()
except Exception as exc:
    print("Cannot open the file:", exc)
```

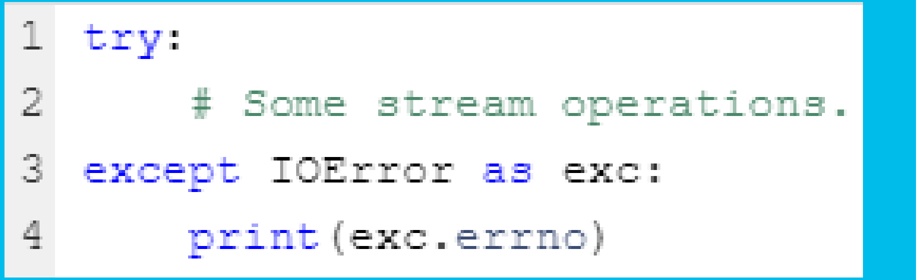
Closing streams

The last operation performed on a stream (**this doesn't include the stdin, stdout, and stderr** streams which don't require it) should be closing.

That action is performed by a method invoked from within open stream object: **stream.close().**

- the name of the function is definitely self-commenting: **close()**
- the function expects exactly no arguments; the stream doesn't need to be opened
- the function returns nothing but raises **IOError** exception in case of error;

Diagnosing stream problems



```
try:
    # Some stream operations.
except IOError as exc:
    print(exc.errno)
```

The **IOError** object is equipped with a property named **errno** (the name comes from the phrase error number) and you can access it as follows:

errno.EACCES → Permission denied

errno.EBADF → Bad file number

errno.EEXIST → File exists

errno.EFBIG → File too large

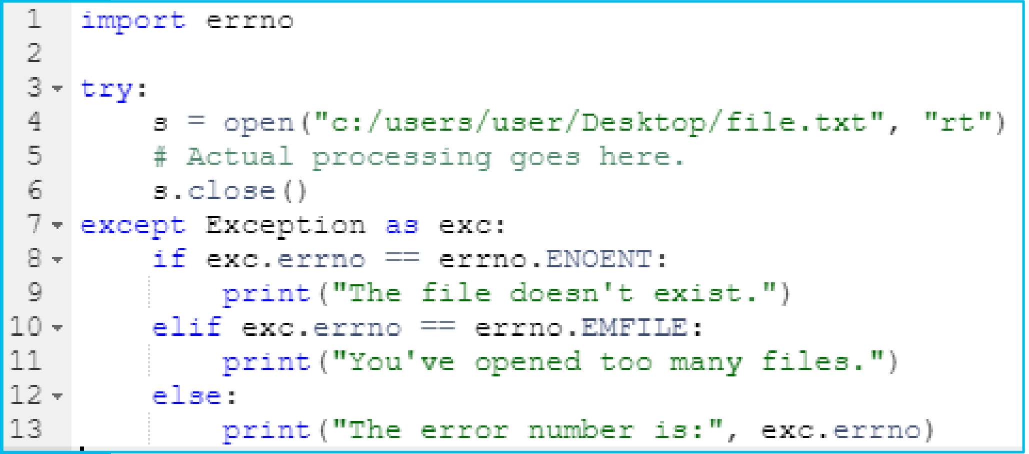
errno.EISDIR → Is a directory

errno.EMFILE → Too many open files

errno.ENOENT → No such file or directory

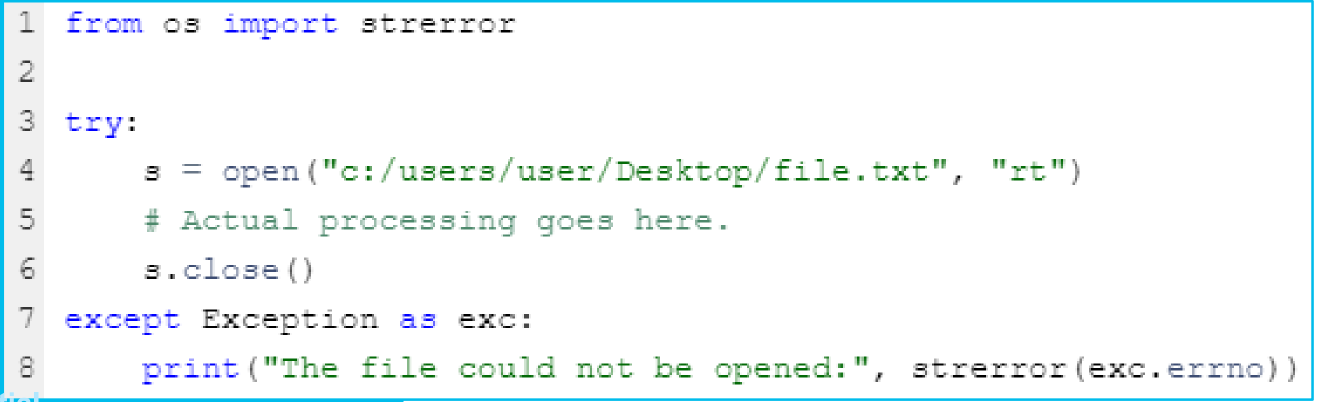
errno.ENOSPC → No space left on device

Diagnosing stream problems: continued



```python
import errno

try:
    s = open("c:/users/user/Desktop/file.txt", "rt")
    # Actual processing goes here.
    s.close()
except Exception as exc:
    if exc.errno == errno.ENOENT:
        print("The file doesn't exist.")
    elif exc.errno == errno.EMFILE:
        print("You've opened too many files.")
    else:
        print("The error number is:", exc.errno)
```

strerror(), and it comes from the **os** module and expects just one argument - an error number.



```python
from os import strerror

try:
    s = open("c:/users/user/Desktop/file.txt", "rt")
    # Actual processing goes here.
    s.close()
except Exception as exc:
    print("The file could not be opened:", strerror(exc.errno))
```

Key takeaways

1. A file needs to be open before it can be processed by a program, and it should be closed when the processing is finished.

Three open modes exist:

- **read mode** – only read operations are allowed;
- **write mode** – only write operations are allowed;
- **update mode** – both writes and reads are allowed.

2. Depending on the physical file content, different Python classes can be used to process files. In general, the BufferedIOBase is able to process any file, while TextIOBase is a specialized class dedicated to processing text files (i.e. files containing human-visible texts divided into lines using new-line markers). Thus, the streams can be divided into binary and text ones.

Key takeaways

3. The following **open()** function syntax is used to open a file:

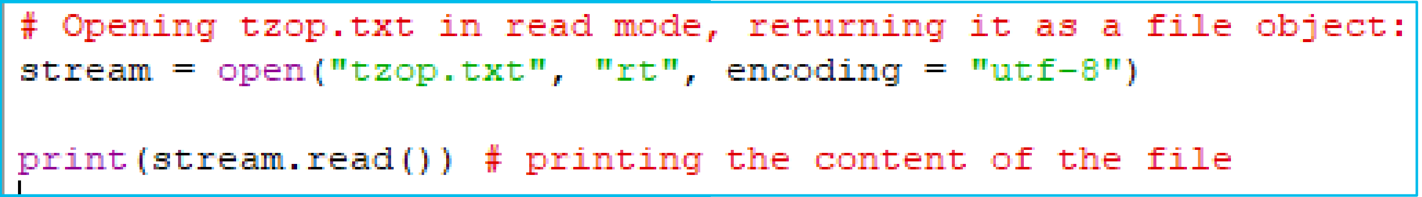
open(*file_name, mode=open_mode, encoding=text_encoding*)

The invocation creates a stream object and associates it with the file named file_name, using the specified open_mode and setting the specified text_encoding, or it raises an exception in the case of an error.

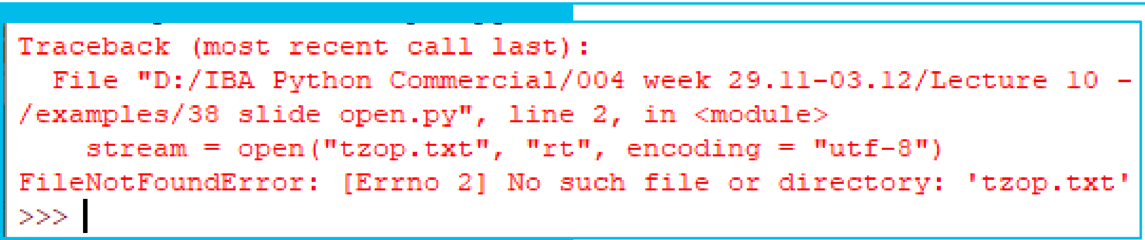
4. Three predefined streams are already open when the program starts:

- **sys.stdin** – standard input;
- **sys.stdout** – standard output;
- **sys.stderr** – standard error output.

4. The **IOError exception object**, created when any file operations fails (including open operations), contains a property named **errno**, which contains the completion code of the failed action. Use this value to diagnose the problem.

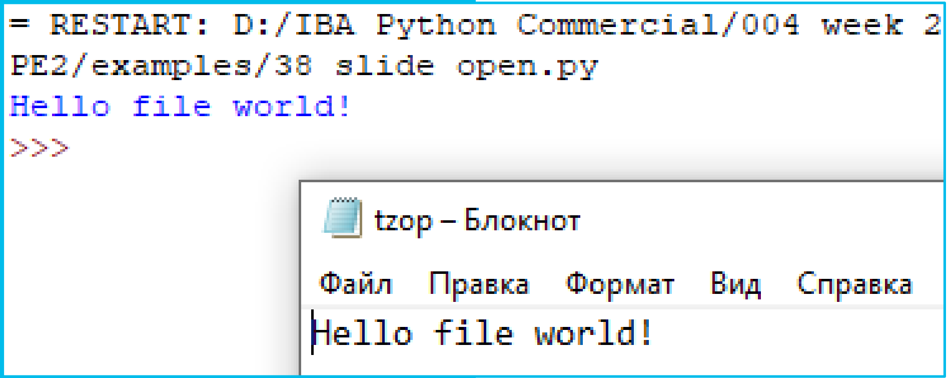


```
# Opening tzop.txt in read mode, returning it as a file object:
stream = open("tzop.txt", "rt", encoding = "utf-8")

print(stream.read()) # printing the content of the file
```



```
Traceback (most recent call last):
  File "D:/IBA Python Commercial/004 week 29.11-03.12/Lecture 10 -
/examples/38 slide open.py", line 2, in <module>
    stream = open("tzop.txt", "rt", encoding = "utf-8")
FileNotFoundError: [Errno 2] No such file or directory: 'tzop.txt'
>>> 
```

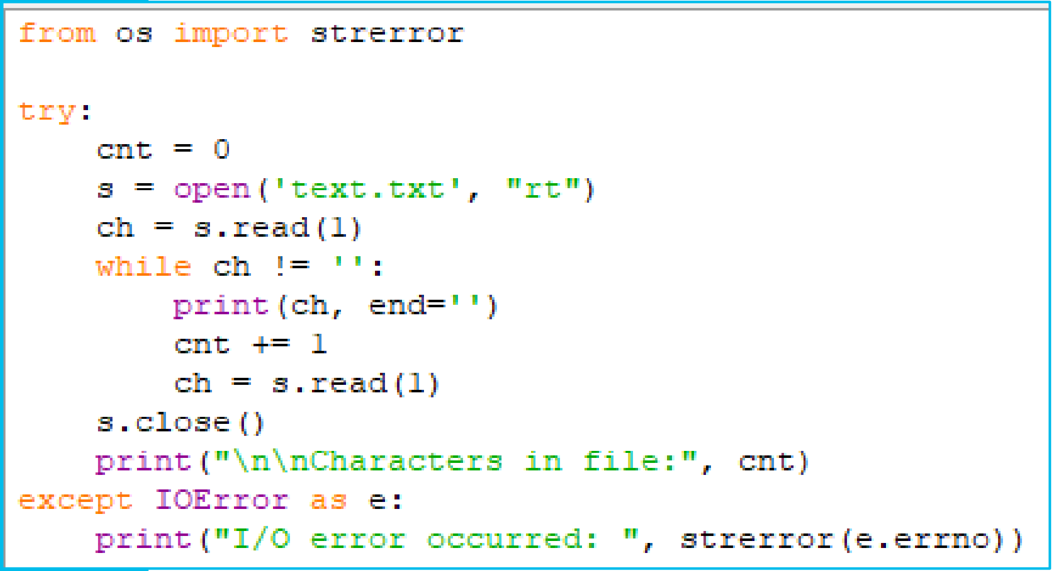
Processing text files



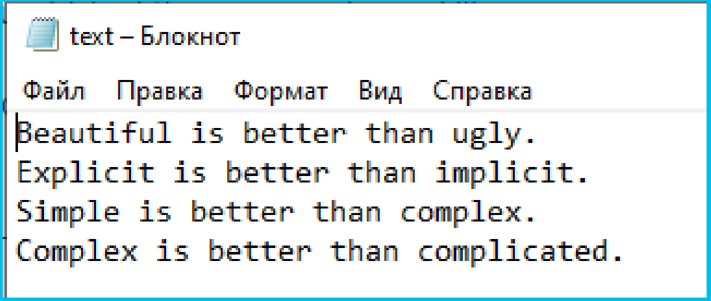
```
= RESTART: D:/IBA Python Commercial/004 week 2
PE2/examples/38 slide open.py
Hello file world!
>>>
```

```
Hello file world!
```

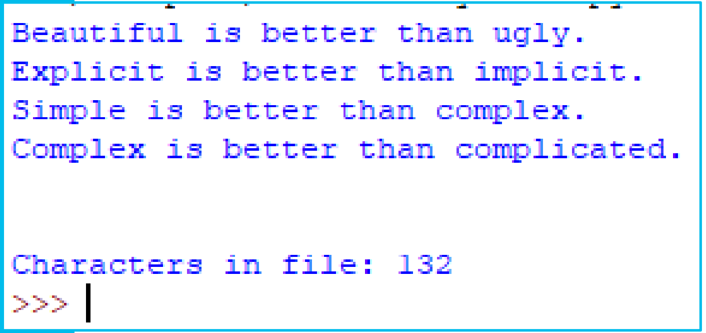
Processing text files: continued



```python
from os import strerror

try:
    cnt = 0
    s = open('text.txt', "rt")
    ch = s.read(1)
    while ch != '':
        print(ch, end='')
        cnt += 1
        ch = s.read(1)
    s.close()
    print("\n\nCharacters in file:", cnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

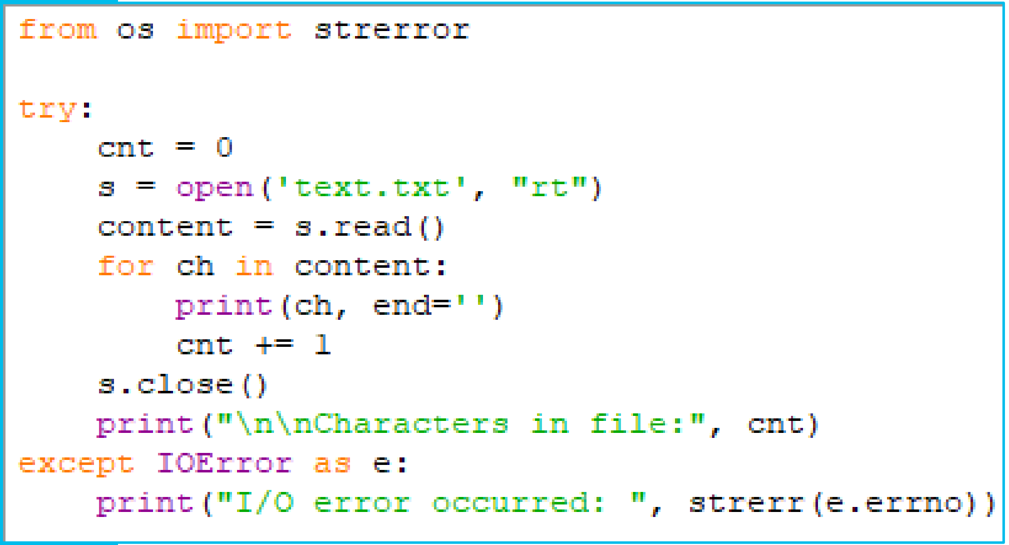


```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
```

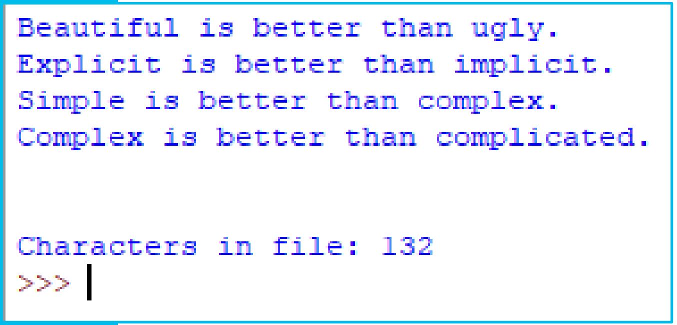


```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.


Characters in file: 132
>>> 
```

Processing text files: continued



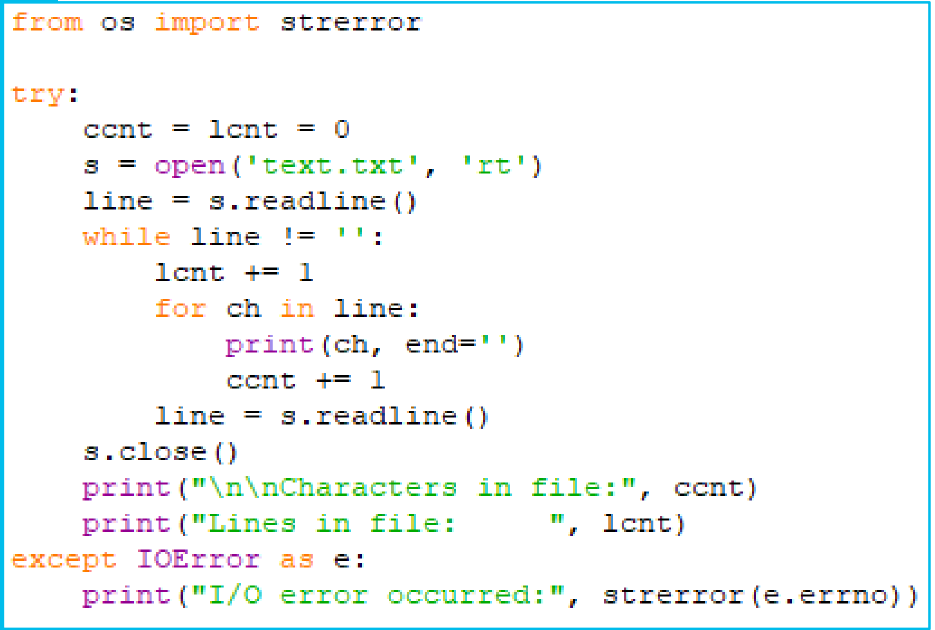
```
from os import strerror

try:
    cnt = 0
    s = open('text.txt', "rt")
    content = s.read()
    for ch in content:
        print(ch, end='')
        cnt += 1
    s.close()
    print("\n\nCharacters in file:", cnt)
except IOError as e:
    print("I/O error occurred: ", strerr(e.errno))
```



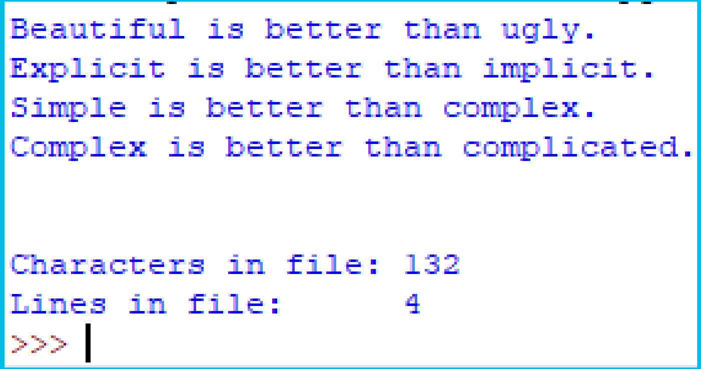
```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.


Characters in file: 132
>>> |
```

Processing text files: **readline()**

The **readline()** method tries to read a complete line of text from the file, and returns it as a string in the case of success. Otherwise, it returns an empty string.

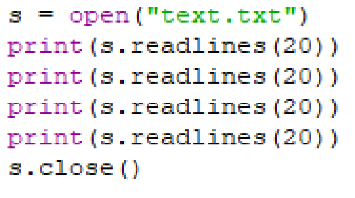


```python
from os import strerror

try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    line = s.readline()
    while line != '':
        lcnt += 1
        for ch in line:
            print(ch, end='')
            ccnt += 1
        line = s.readline()
    s.close()
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:     ", lcnt)
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

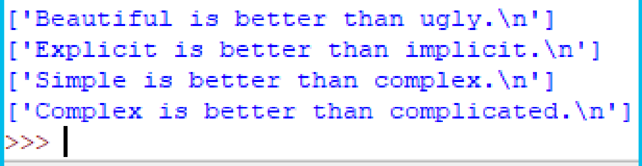


```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.


Characters in file: 132
Lines in file:      4
>>> 
```

Processing text files: **readlines()**



```
s = open("text.txt")
print(s.readlines(20))
print(s.readlines(20))
print(s.readlines(20))
print(s.readlines(20))
s.close()
```

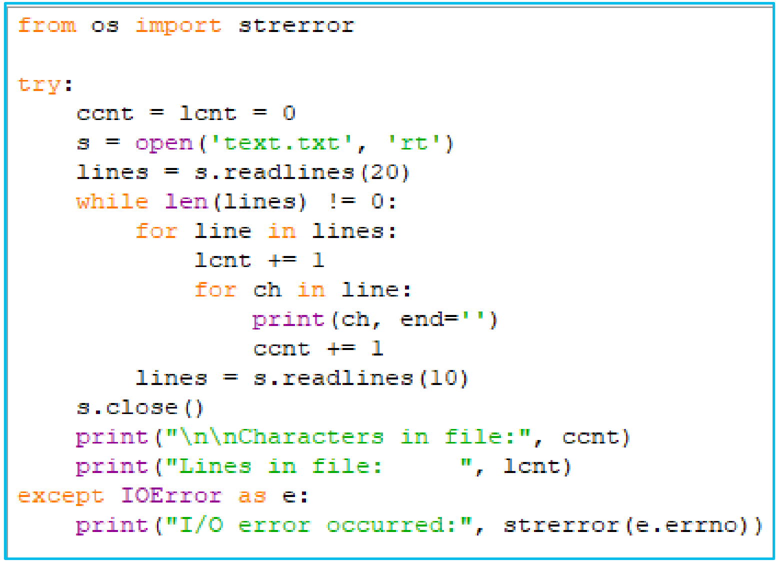


```
['Beautiful is better than ugly.\n']
['Explicit is better than implicit.\n']
['Simple is better than complex.\n']
['Complex is better than complicated.\n']
>>> 
```

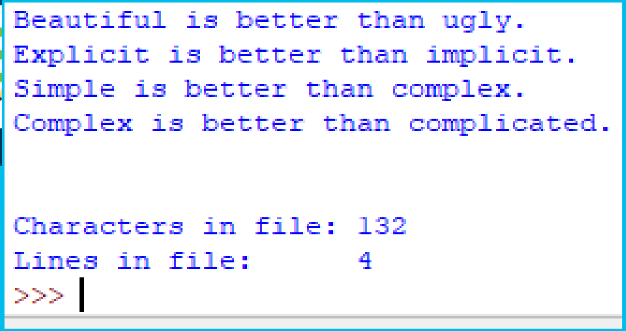
The **readlines()** method, when invoked without arguments, tries to read all the file contents, and returns a list of strings, one element per file line.

The maximum accepted input buffer size is passed to the method as its argument.

Note: when there is nothing to read from the file, the method returns an empty list. Use it to detect the end of the file.



```
from os import strerror

try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    lines = s.readlines(20)
    while len(lines) != 0:
        for line in lines:
            lcnt += 1
            for ch in line:
                print(ch, end='')
                ccnt += 1
        lines = s.readlines(10)
    s.close()
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:     ", lcnt)
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

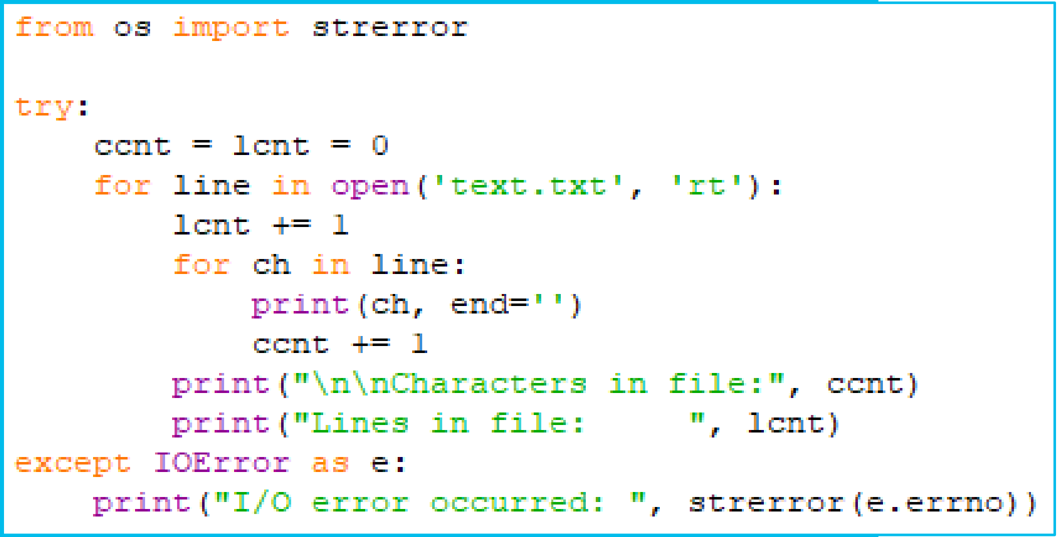


```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.


Characters in file: 132
Lines in file:      4
>>> 
```

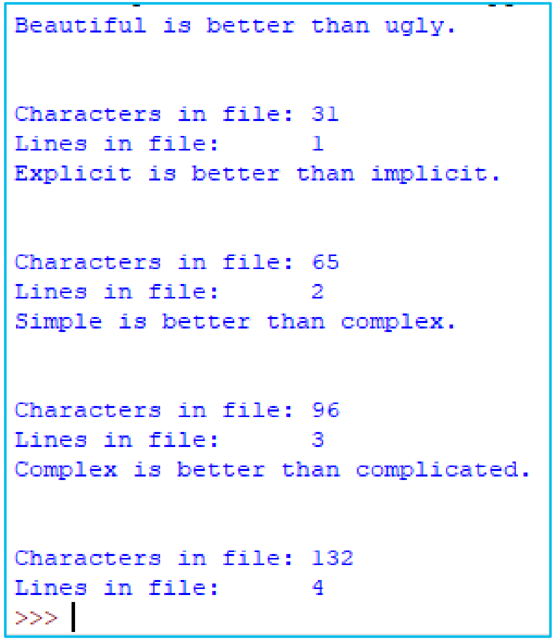
Processing text files: continued

The iteration protocol defined for the file object is very simple - its __next__ method just returns the next line read in from the file.

Moreover, you can expect that the object automatically invokes **close()** when any of the file reads reaches the end of the file.



```
from os import strerror

try:
    ccnt = lcnt = 0
    for line in open('text.txt', 'rt'):
        lcnt += 1
        for ch in line:
            print(ch, end='')
            ccnt += 1
        print("\n\nCharacters in file:", ccnt)
        print("Lines in file:      ", lcnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

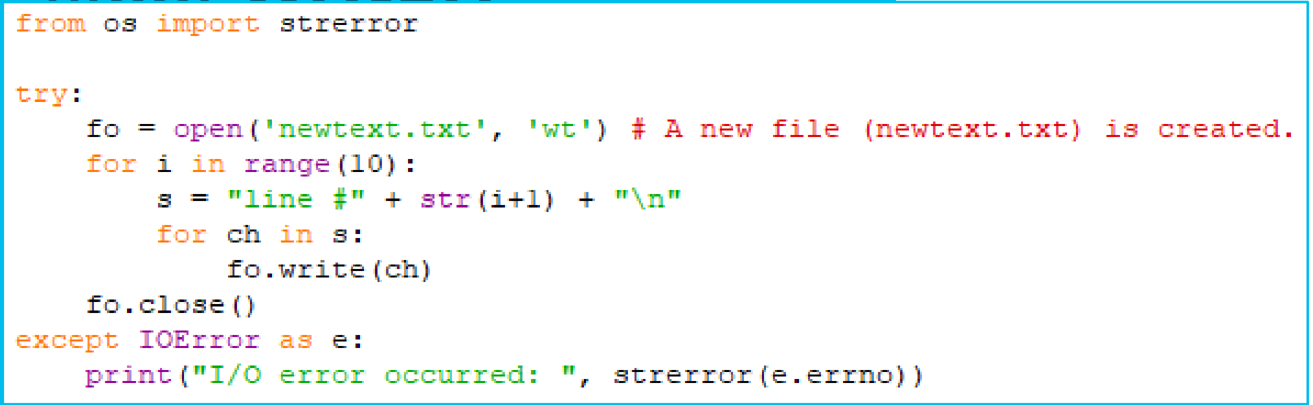


```
Beautiful is better than ugly.


Characters in file: 31
Lines in file:      1
Explicit is better than implicit.


Characters in file: 65
Lines in file:      2
Simple is better than complex.


Characters in file: 96
Lines in file:      3
Complex is better than complicated.


Characters in file: 132
Lines in file:      4
>>> 
```

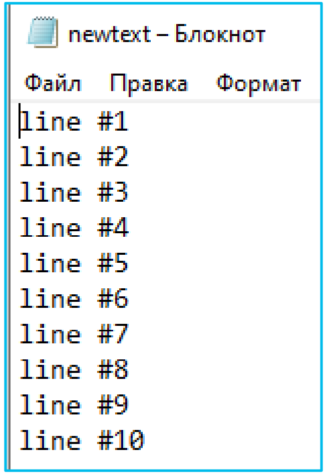
Dealing with text files: **write()**

The method is named **write()** and it expects just one argument - a string that will be transferred to an open file (don't forget - the open mode should reflect the way in which the data is transferred - writing a file opened in read mode won't succeed).

No newline character is added to the write()'s argument, so you have to add it yourself if you want the file to be filled with a number of lines.

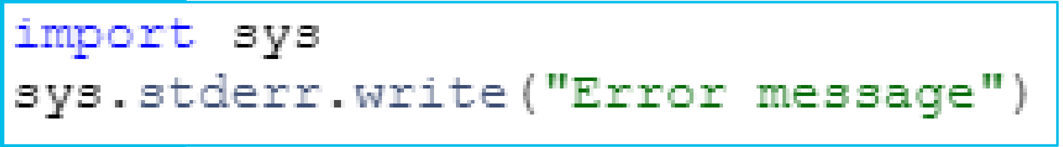


```
from os import strerror

try:
    fo = open('newtext.txt', 'wt') # A new file (newtext.txt) is created.
    for i in range(10):
        s = "line #" + str(i+1) + "\n"
        for ch in s:
            fo.write(ch)
    fo.close()
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

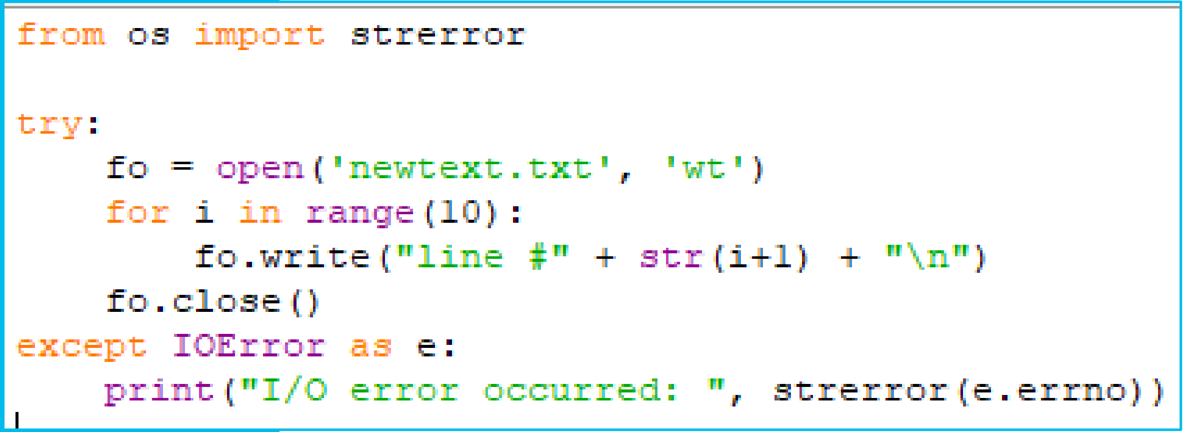


```
line #1
line #2
line #3
line #4
line #5
line #6
line #7
line #8
line #9
line #10
```

Dealing with text files: continued



```
import sys
sys.stderr.write("Error message")
```



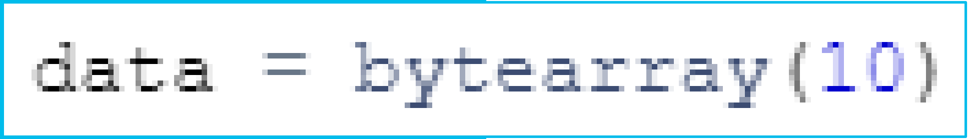
```
from os import strerror

try:
    fo = open('newtext.txt', 'wt')
    for i in range(10):
        fo.write("line #" + str(i+1) + "\n")
    fo.close()
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

What is a bytearray?

Amorphous data is data which have no specific shape or form - they are just a series of bytes.

Python has more than one such container - one of them is a specialized class name **bytearray** - as the name suggests, it's an array containing (amorphous) bytes.

Note: such a constructor fills the whole array with zeros.

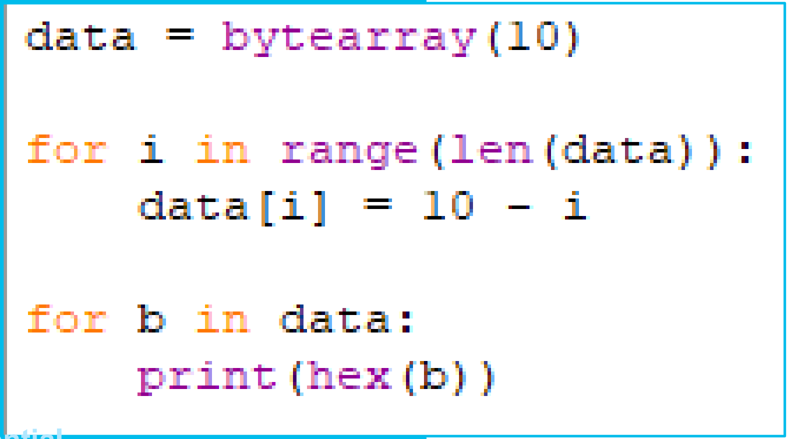


```
data = bytearray(10)
```

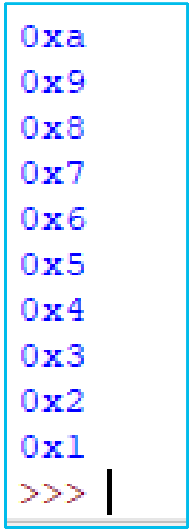
Bytearrays: continued

Bytearrays resemble lists in many respects. For example, they are **mutable**, they're a subject of the **len()** function, and you can access any of their elements using conventional **indexing.**

There is one important limitation - you mustn't set any byte array elements with a value which is not an integer (violating this rule will cause a TypeError exception) and you're not allowed to assign a value that doesn't come from the range 0 to 255 inclusive (unless you want to provoke a ValueError exception).

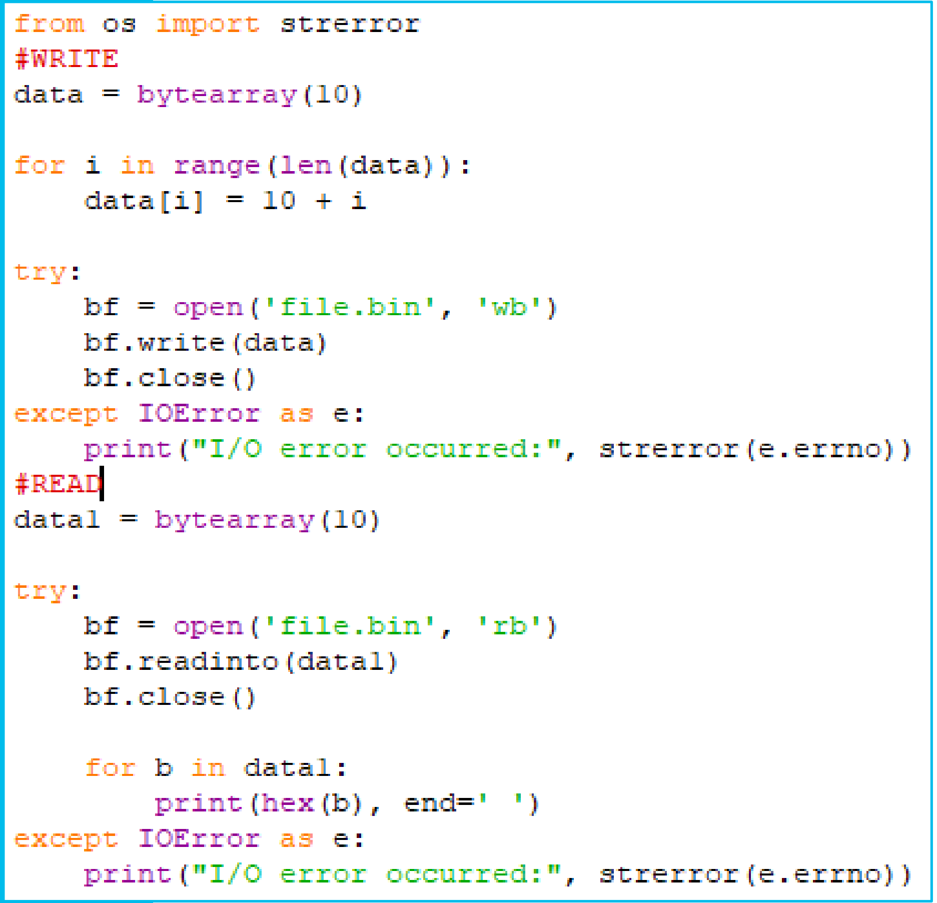


```
data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 - i

for b in data:
    print(hex(b))
```

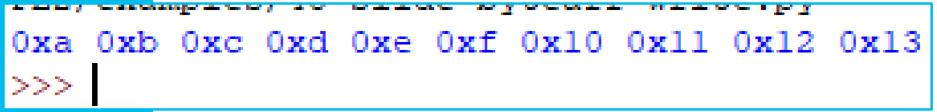


```
0xa
0x9
0x8
0x7
0x6
0x5
0x4
0x3
0x2
0x1
>>> 
```

Bytearrays: write read



```
from os import strerror
#WRITE
data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 + i

try:
    bf = open('file.bin', 'wb')
    bf.write(data)
    bf.close()
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
#READ
data1 = bytearray(10)

try:
    bf = open('file.bin', 'rb')
    bf.readinto(data1)
    bf.close()

    for b in data1:
        print(hex(b), end=' ')
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

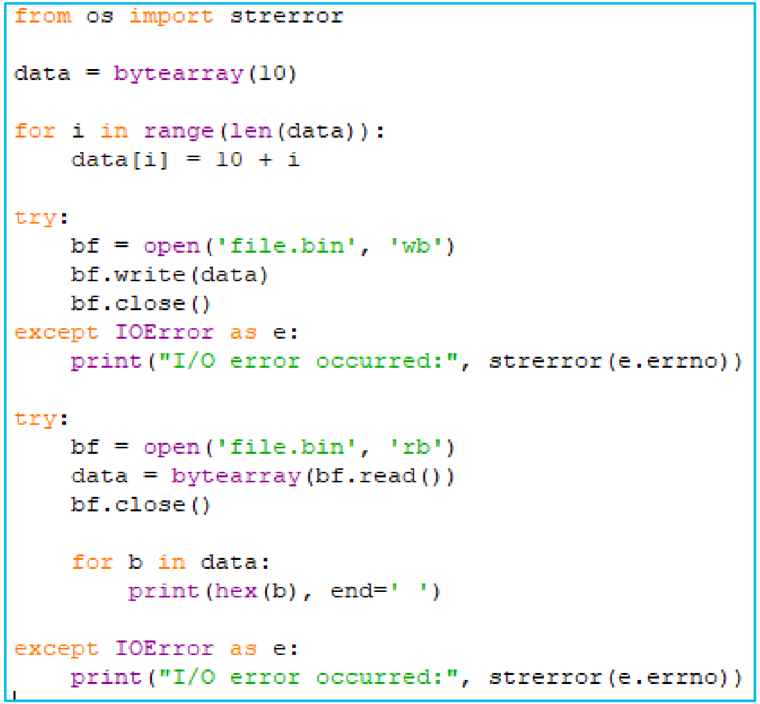


```
0xa 0xb 0xc 0xd 0xe 0xf 0x10 0x11 0x12 0x13
>>> 
```

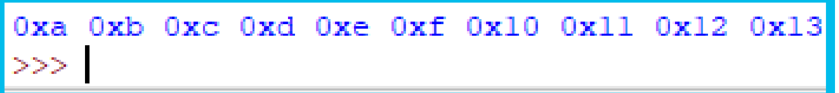
How to read bytes from a stream

read() invoked without arguments, it tries to read all the contents of the file into the memory, making them a part of a newly created object of the bytes class.

This class has some **similarities to bytearray**, with the exception of one significant difference - it's **immutable.**



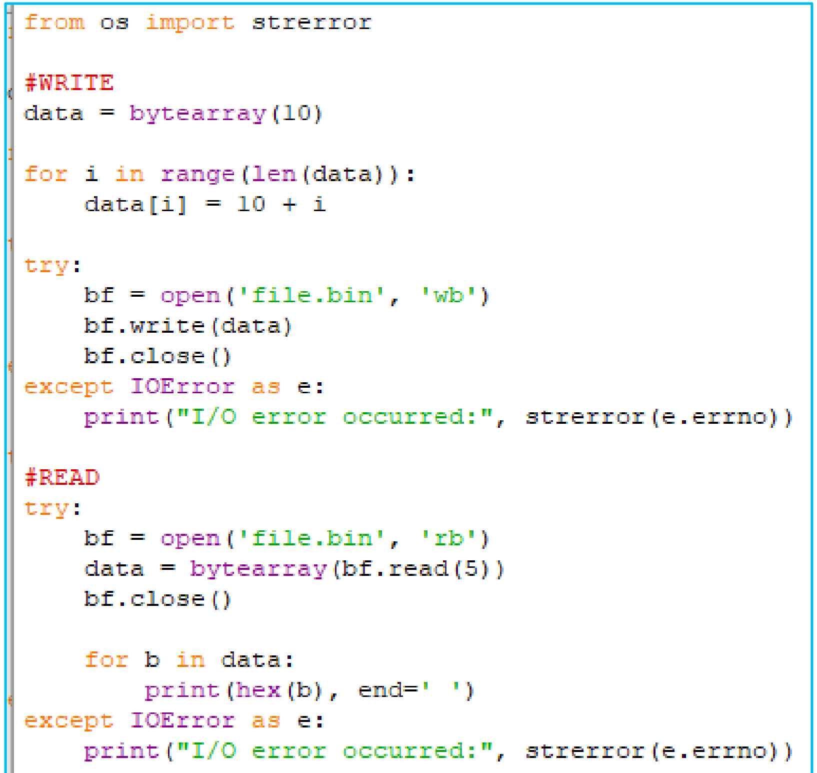
```
from os import strerror

data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 + i

try:
    bf = open('file.bin', 'wb')
    bf.write(data)
    bf.close()
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))

try:
    bf = open('file.bin', 'rb')
    data = bytearray(bf.read())
    bf.close()

    for b in data:
        print(hex(b), end=' ')

except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```



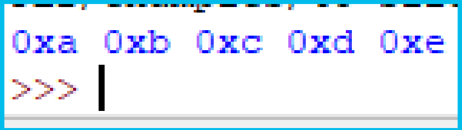
```
0xa 0xb 0xc 0xd 0xe 0xf 0x10 0x11 0x12 0x13
>>> 
```

How to read bytes from a stream: continued

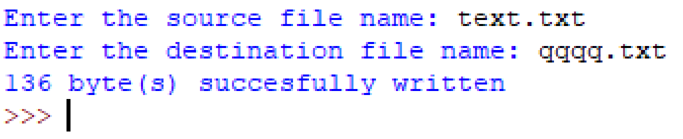
If the **read()** method is invoked with an argument, it specifies the maximum number of bytes to be read.



```
from os import strerror

#WRITE
data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 + i

try:
    bf = open('file.bin', 'wb')
    bf.write(data)
    bf.close()
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))

#READ
try:
    bf = open('file.bin', 'rb')
    data = bytearray(bf.read(5))
    bf.close()

    for b in data:
        print(hex(b), end=' ')
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

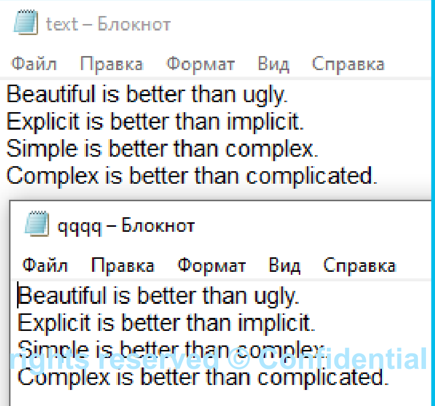


```
0xa 0xb 0xc 0xd 0xe
>>>
```



```
Enter the source file name: text.txt
Enter the destination file name: qqqq.txt
136 byte(s) succesfully written
>>> 
```

Copying files - a simple and functional tool

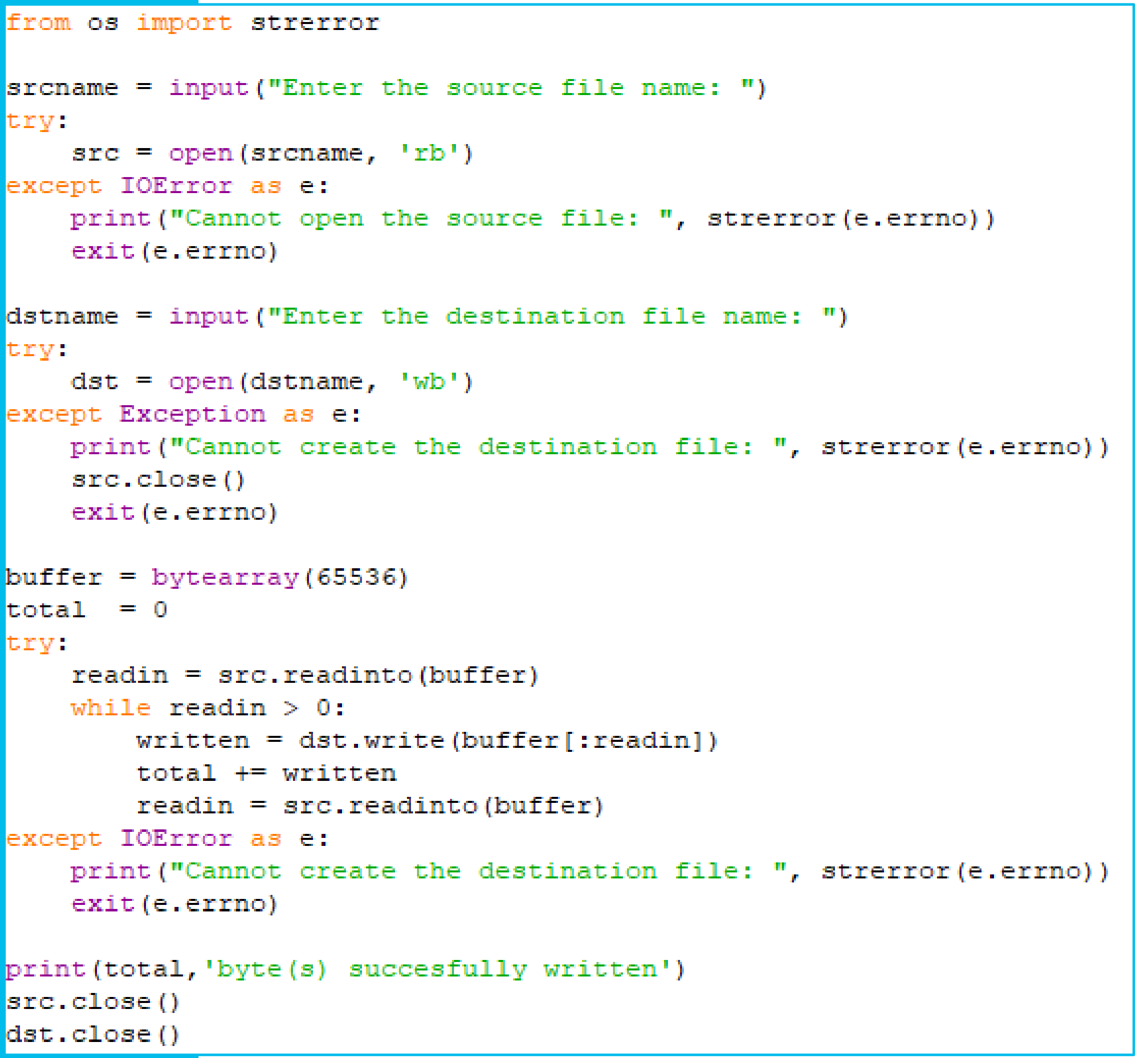


text – Блокнот
Файл Правка Формат Вид Справка

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.

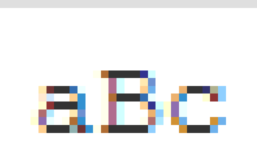
qqqq – Блокнот
Файл Правка Формат Вид Справка

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.

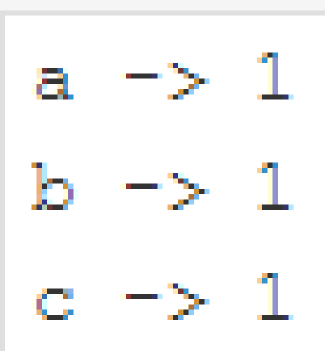


```python
from os import strerror

srcname = input("Enter the source file name: ")
try:
    src = open(srcname, 'rb')
except IOError as e:
    print("Cannot open the source file: ", strerror(e.errno))
    exit(e.errno)

dstname = input("Enter the destination file name: ")
try:
    dst = open(dstname, 'wb')
except Exception as e:
    print("Cannot create the destination file: ", strerror(e.errno))
    src.close()
    exit(e.errno)

buffer = bytearray(65536)
total  = 0
try:
    readin = src.readinto(buffer)
    while readin > 0:
        written = dst.write(buffer[:readin])
        total += written
        readin = src.readinto(buffer)
except IOError as e:
    print("Cannot create the destination file: ", strerror(e.errno))
    exit(e.errno)

print(total,'byte(s) succesfully written')
src.close()
dst.close()
```

Home work 11_1
lower(), isalpha()
dict.keys()



```
aBc
```



```
a -> 1
b -> 1
c -> 1
```

A text file contains some text (nothing unusual) but we need to know how often (or how rare) each letter appears in the text. Such an analysis may be useful in cryptography, so we want to be able to do that in reference to the Latin alphabet.

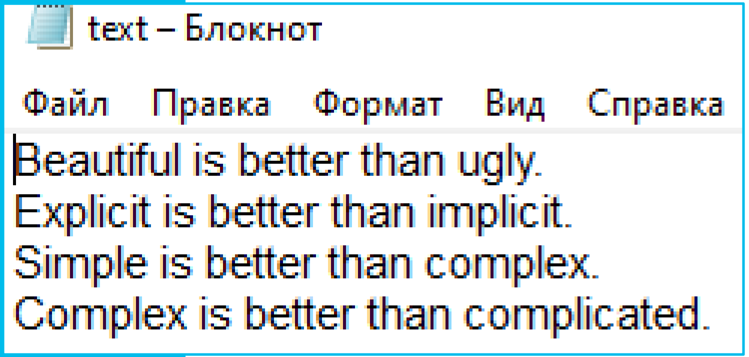
Your task is to write a program which:

- **asks the user for the input file's name;**
- **reads the file (if possible)** and **counts all the Latin letters (lower- and upper-case letters are treated as equal)**
- **prints a simple histogram in alphabetical order** (*only non-zero counts should be presented*)

Create a test file for the code, and check if your histogram contains valid results.

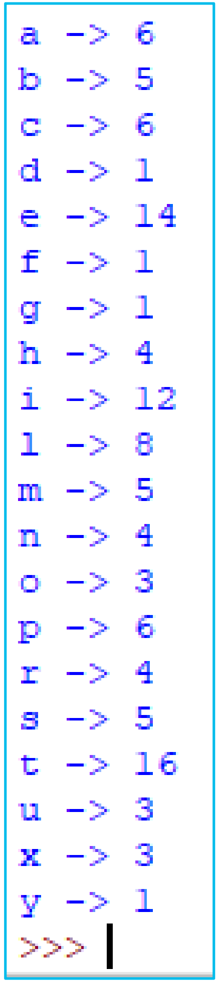
Assuming that the test file contains just one line filled with:

Home work 11_1 lower(), isalpha() dict.keys()



```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
```

```
counters = {chr(ch): 0 for ch in range(ord('a'), ord('z') + 1)}

print(counters)
```



```
a -> 6
b -> 5
c -> 6
d -> 1
e -> 14
f -> 1
g -> 1
h -> 4
i -> 12
l -> 8
m -> 5
n -> 4
o -> 3
p -> 6
r -> 4
s -> 5
t -> 16
u -> 3
x -> 3
y -> 1
>>> 
```

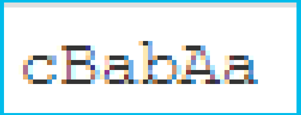
Home work 11_1_1*

The previous code needs to be improved. It's okay, but it has to be better.

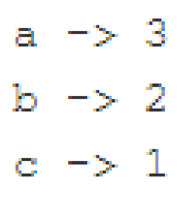
Your task is to make some amendments, which generate the following results:

- **the output point-histogram will be sorted based on the characters' frequency (the bigger counter should be presented first)**
- **the histogram should be sent to a file with the same name as the input one, but with the suffix '.hist' (it should be concatenated to the original name)**

Assuming that the input file contains just one line filled with:



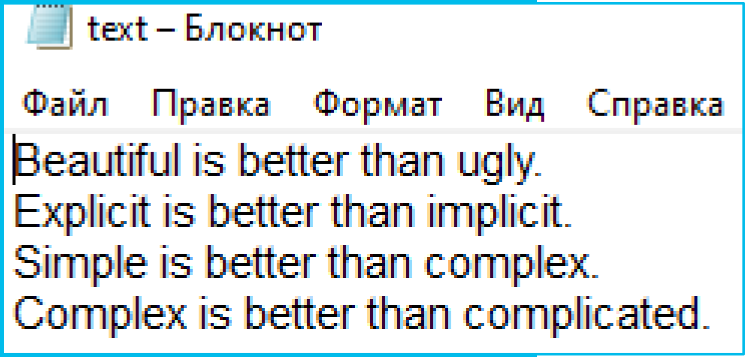
```
cBabAa
```



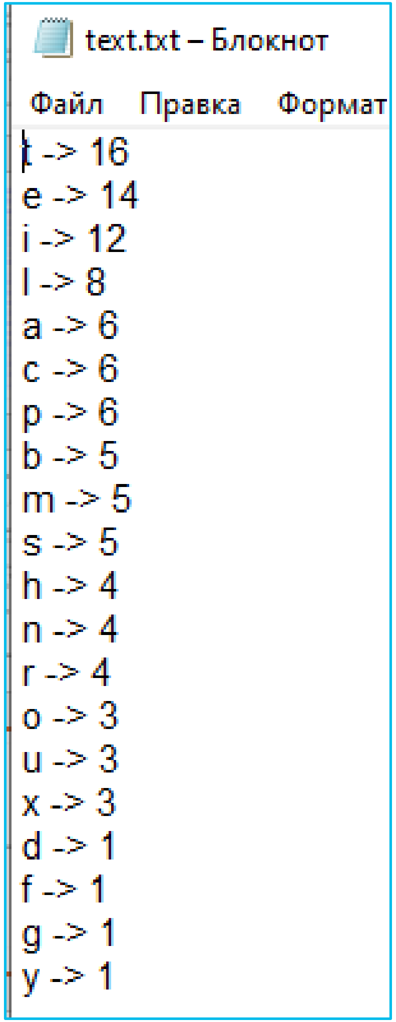
```
a -> 3
b -> 2
c -> 1
```

Tip: try to use a lambda to change the sort order.

Home work 11_1_1* sorted()



```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
```



```
t -> 16
e -> 14
i -> 12
l -> 8
a -> 6
c -> 6
p -> 6
b -> 5
m -> 5
s -> 5
h -> 4
n -> 4
r -> 4
o -> 3
u -> 3
x -> 3
d -> 1
f -> 1
g -> 1
y -> 1
```

Home work 11_2 Student

f.write(fname + ' ' +)

Prof. Stefan conducts classes with students and regularly makes notes in a text file. Each line of the file contains three elements: the student's first name, the student's last name, and the number of point the student received during certain classes.

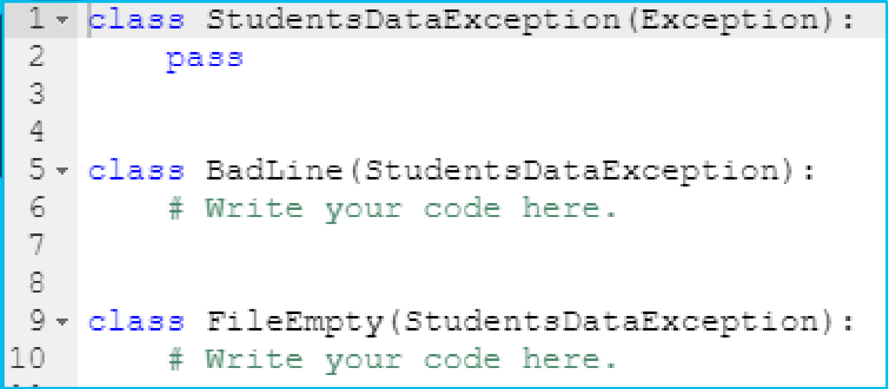
The elements are separated with white spaces. Each student may appear more than once inside Prof. Terentiev's file.

The file may look as follows:

student_data.txt

Your task is to write a program which:

- **asks the user for Prof. Stefan's file name**;
- **reads the file contents and counts the sum of the received points for each student**;
- **prints a simple (but sorted) report (next slide)**



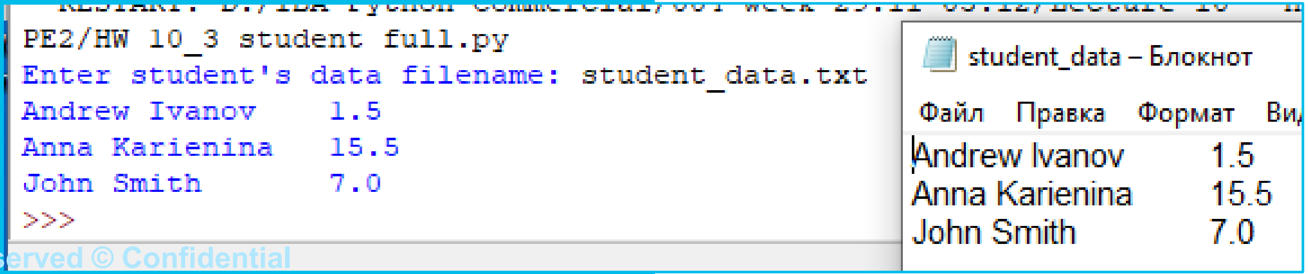
```python
class StudentsDataException(Exception):
    pass


class BadLine(StudentsDataException):
    # Write your code here.


class FileEmpty(StudentsDataException):
    # Write your code here.
```

Home work 11_2 Student

Note:

- your program must be fully protected against all possible failures: **the file's non-existence, the file's emptiness, or any input data failures**; encountering any data error should cause immediate program termination, and the erroneous should be presented to the user;
- implement and use your own exceptions hierarchy - we've presented it in the editor; the second exception should be raised when a bad line is detect, and the third when the source file exists but is empty.

Tip: Use a dictionary to store the students' data.

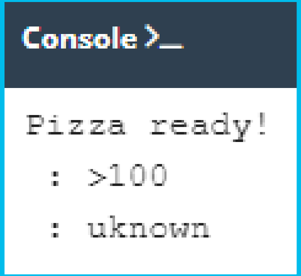


```
 RESTART: D:/IBA Python Commercial/001 Week 29.11 03.12/Lecture 10
PE2/HW 10_3 student full.py
Enter student's data filename: student_data.txt
Andrew Ivanov     1.5
Anna Karienina    15.5
John Smith        7.0
>>>
```

```
Andrew Ivanov       1.5
Anna Karienina      15.5
John Smith          7.0
```

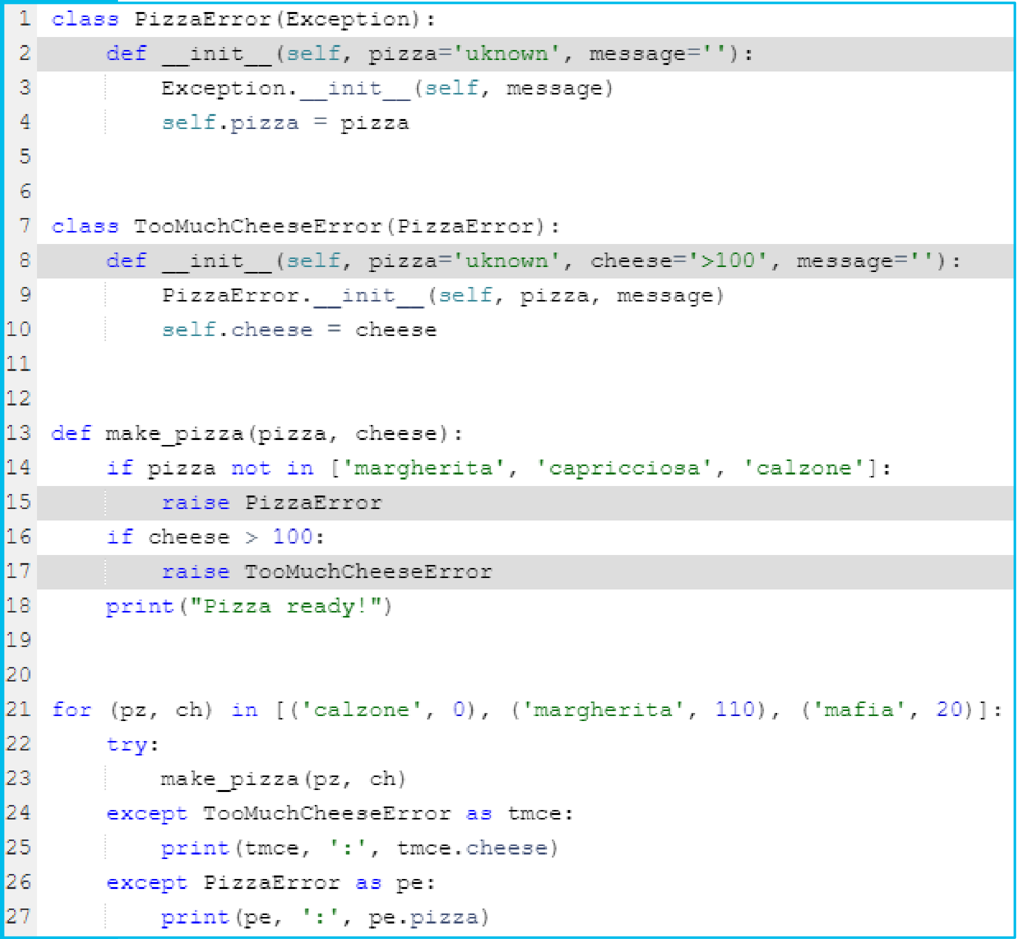
Pizzeria

Home work 11_3 Pizzeria



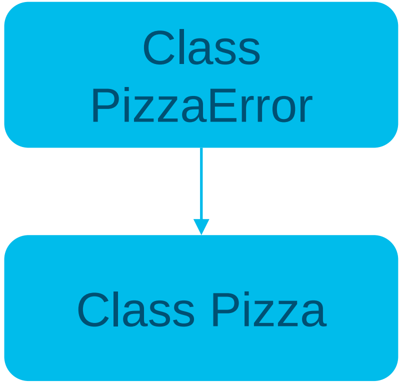
Console >_

```
Pizza ready!
 : >100
 : uknown
```



```python
class PizzaError(Exception):
    def __init__(self, pizza='uknown', message=''):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza='uknown', cheese='>100', message=''):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese


def make_pizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError
    if cheese > 100:
        raise TooMuchCheeseError
    print("Pizza ready!")


for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
    try:
        make_pizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

- Для Обработки исключений
- Возникает при добавлении пиццы в меню – в случае если такая пицца там уже есть

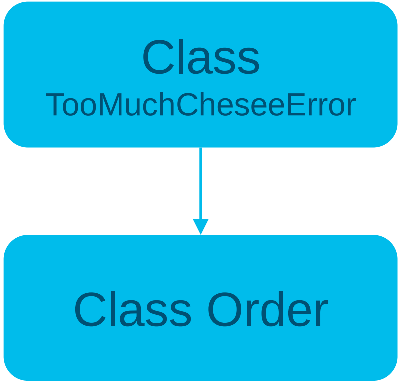


Class PizzaError

Class Pizza

- Для хранения списка доступных пицц
- Возможность добавить новые пиццы в меню
- Возможность удалить из меню те пиццы которые мы больше не готовим

- Для Обработки исключений
- Возникает при создании заказа – в случае если сыра больше 100



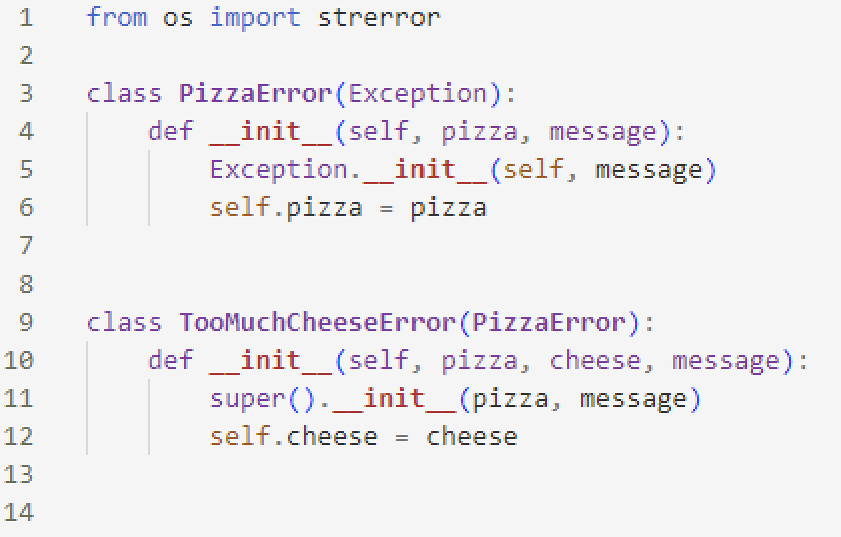
Class TooMuchCheseeError

Class Order

- Для обработки заказов
- Возможность сделать пиццы по заказу
- Возможность записать выполненный заказ в файл

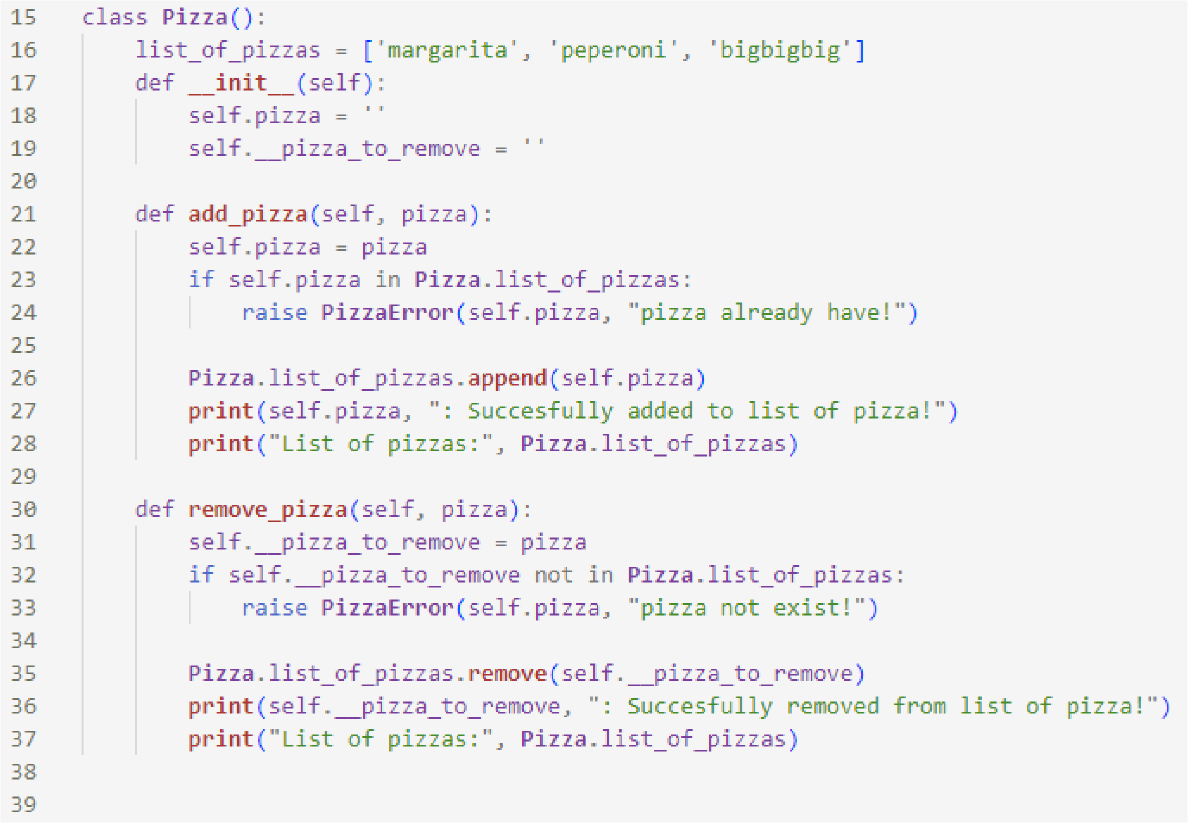
Pizza – parent
Order - child

Exceptions



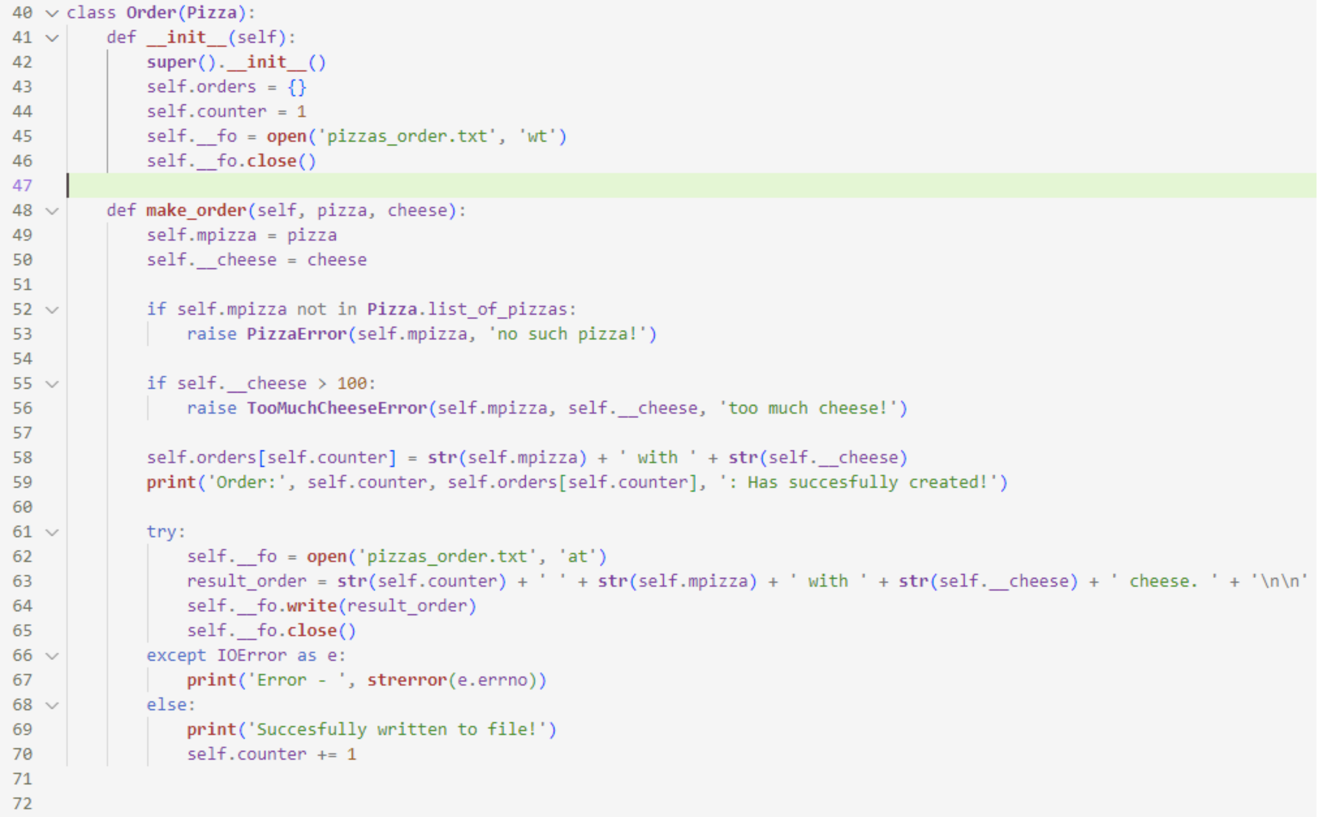
```python
from os import strerror

class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        super().__init__(pizza, message)
        self.cheese = cheese
```

Pizza



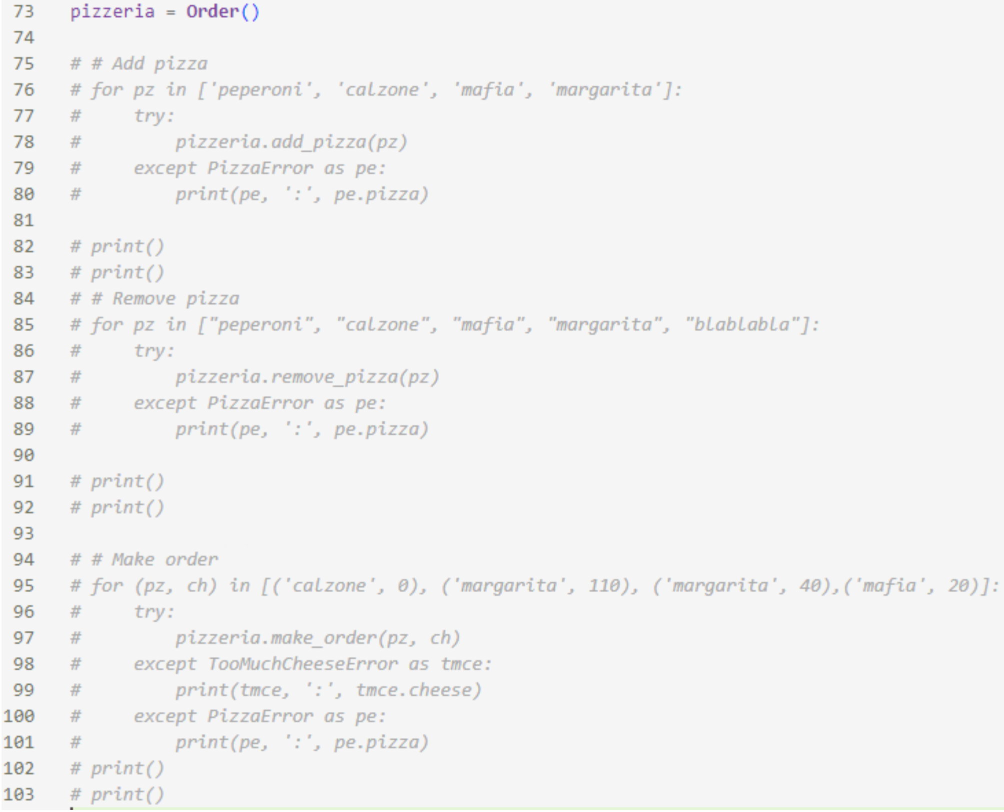
```python
class Pizza():
    list_of_pizzas = ['margarita', 'peperoni', 'bigbigbig']
    def __init__(self):
        self.pizza = ''
        self.__pizza_to_remove = ''

    def add_pizza(self, pizza):
        self.pizza = pizza
        if self.pizza in Pizza.list_of_pizzas:
            raise PizzaError(self.pizza, "pizza already have!")

        Pizza.list_of_pizzas.append(self.pizza)
        print(self.pizza, ": Succesfully added to list of pizza!")
        print("List of pizzas:", Pizza.list_of_pizzas)

    def remove_pizza(self, pizza):
        self.__pizza_to_remove = pizza
        if self.__pizza_to_remove not in Pizza.list_of_pizzas:
            raise PizzaError(self.pizza, "pizza not exist!")

        Pizza.list_of_pizzas.remove(self.__pizza_to_remove)
        print(self.__pizza_to_remove, ": Succesfully removed from list of pizza!")
        print("List of pizzas:", Pizza.list_of_pizzas)


```

Order

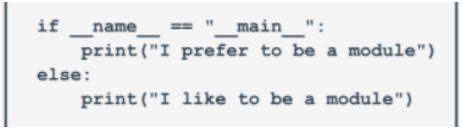


```
class Order(Pizza):
    def __init__(self):
        super().__init__()
        self.orders = {}
        self.counter = 1
        self.__fo = open('pizzas_order.txt', 'wt')
        self.__fo.close()

    def make_order(self, pizza, cheese):
        self.mpizza = pizza
        self.__cheese = cheese

        if self.mpizza not in Pizza.list_of_pizzas:
            raise PizzaError(self.mpizza, 'no such pizza!')

        if self.__cheese > 100:
            raise TooMuchCheeseError(self.mpizza, self.__cheese, 'too much cheese!')

        self.orders[self.counter] = str(self.mpizza) + ' with ' + str(self.__cheese)
        print('Order:', self.counter, self.orders[self.counter], ': Has succesfully created!')

        try:
            self.__fo = open('pizzas_order.txt', 'at')
            result_order = str(self.counter) + ' ' + str(self.mpizza) + ' with ' + str(self.__cheese) + ' cheese. ' + '\n\n'
            self.__fo.write(result_order)
            self.__fo.close()
        except IOError as e:
            print('Error - ', strerror(e.errno))
        else:
            print('Succesfully written to file!')
            self.counter += 1
```

Checker



```
pizzeria = Order()

# # Add pizza
# for pz in ['peperoni', 'calzone', 'mafia', 'margarita']:
#     try:
#         pizzeria.add_pizza(pz)
#     except PizzaError as pe:
#         print(pe, ':', pe.pizza)

# print()
# print()
# # Remove pizza
# for pz in ["peperoni", "calzone", "mafia", "margarita", "blablabla"]:
#     try:
#         pizzeria.remove_pizza(pz)
#     except PizzaError as pe:
#         print(pe, ':', pe.pizza)

# print()
# print()

# # Make order
# for (pz, ch) in [('calzone', 0), ('margarita', 110), ('margarita', 40),('mafia', 20)]:
#     try:
#         pizzeria.make_order(pz, ch)
#     except TooMuchCheeseError as tmce:
#         print(tmce, ':', tmce.cheese)
#     except PizzaError as pe:
#         print(pe, ':', pe.pizza)
# print()
# print()
```



```
if __name__ == "__main__":
    print("I prefer to be a module")
else:
    print("I like to be a module")
```

← At the end of each file

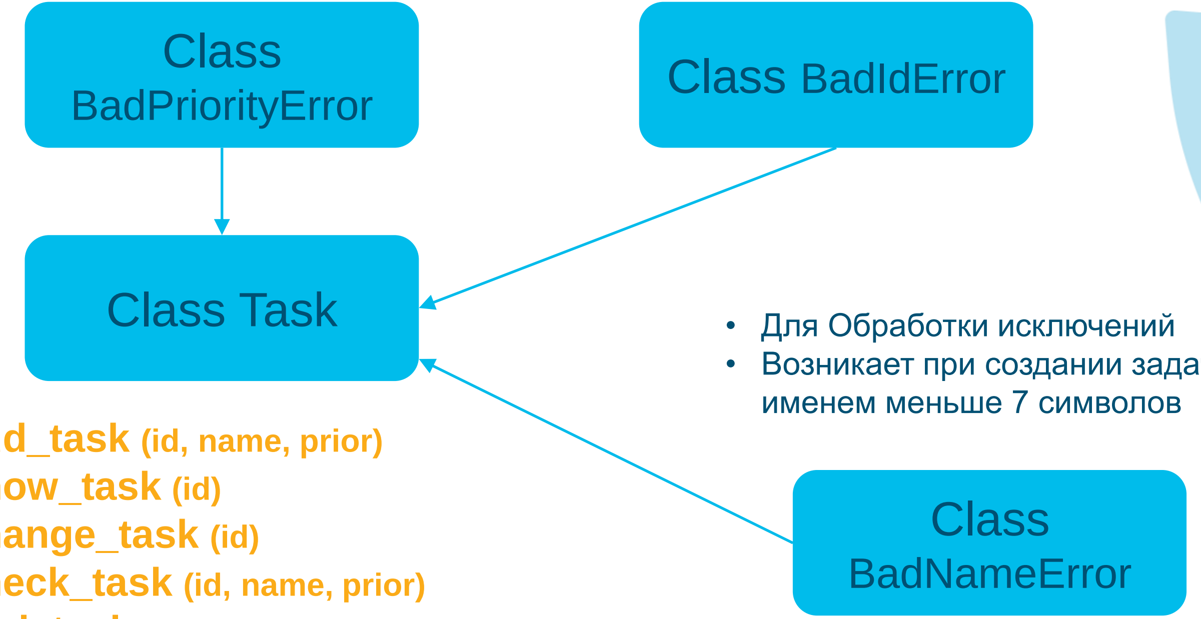
Home work 11_3
Pizzeria

*Step by step video with live coding
*I'll send you after the lesson
*length estimated 45min

ToDo list

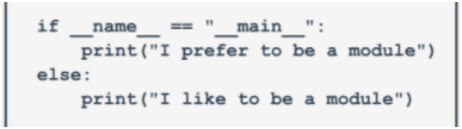
- Для Обработки исключений
- Возникает при создании задачи с приоритетом меньше 1 и/или больше 100

- Для Обработки исключений
- Возникает при создании задачи с id меньше 1 и/или больше 15



- Для Обработки исключений
- Возникает при создании задачи с именем меньше 7 символов

add_task (id, name, prior)
show_task (id)
change_task (id)
check_task (id, name, prior)
find_task (by id)
write_to_file



```
if __name__ == "__main__":
    print("I prefer to be a module")
else:
    print("I like to be a module")
```

← At the end of each file

Home work 11_4
To do list

Create the structure of classes for the todo project.

class Task:

- add_task (id, name, prior)
- show_task (id)
- change_task (id)
- check_task (id, name, prior)
- find_task (by id)
- write_to_file

Classes for exceptions:

- BadPriorityError
- BadIdError
- BadNameError

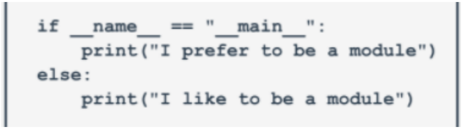
!Add the main function!

Home works structure

Try to use packages

My_projects:

- **modules**
- **packages**
- **main.py**



```
if __name__ == "__main__":
    print("I prefer to be a module")
else:
    print("I like to be a module")
```

← At the end of each file

The Python Standard Library

https://docs.python.org/3/library/

https://docs.python.org/3/library/os.html

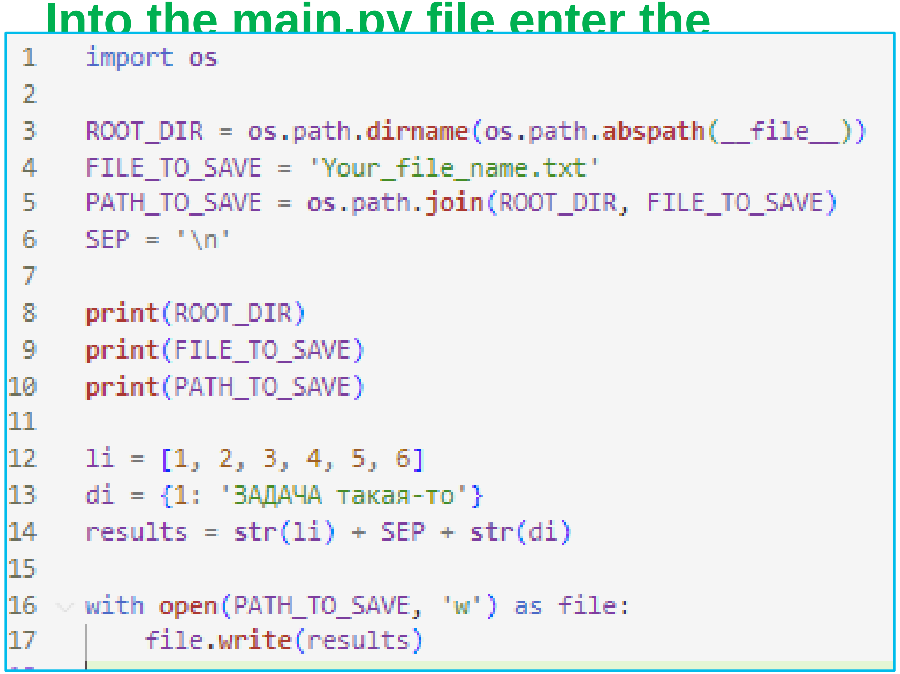
https://docs.python.org/3/library/calendar.html

https://docs.python.org/3/library/datetime.html

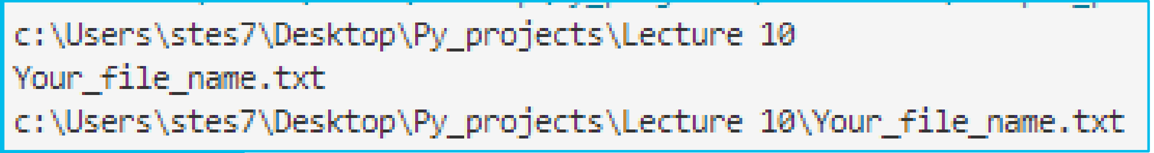
Into the main.py file enter the

HINT

How to save without of using the absolute path?



```python
import os

ROOT_DIR = os.path.dirname(os.path.abspath(__file__))
FILE_TO_SAVE = 'Your_file_name.txt'
PATH_TO_SAVE = os.path.join(ROOT_DIR, FILE_TO_SAVE)
SEP = '\n'

print(ROOT_DIR)
print(FILE_TO_SAVE)
print(PATH_TO_SAVE)

li = [1, 2, 3, 4, 5, 6]
di = {1: 'ЗАДАЧА такая-то'}
results = str(li) + SEP + str(di)

with open(PATH_TO_SAVE, 'w') as file:
    file.write(results)
```

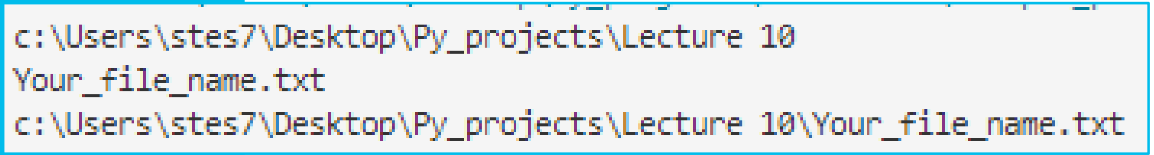


```
c:\Users\stes7\Desktop\Py_projects\Lecture 10
Your_file_name.txt
c:\Users\stes7\Desktop\Py_projects\Lecture 10\Your_file_name.txt
```

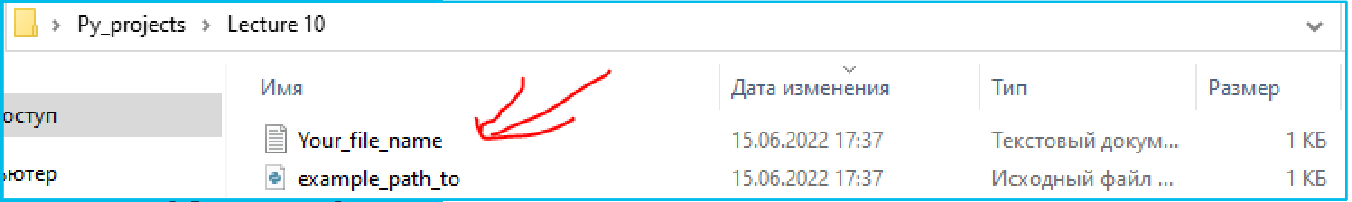
https://stackoverflow.com/questions/25389095/python-get-path-of-root-project-structure

HINT

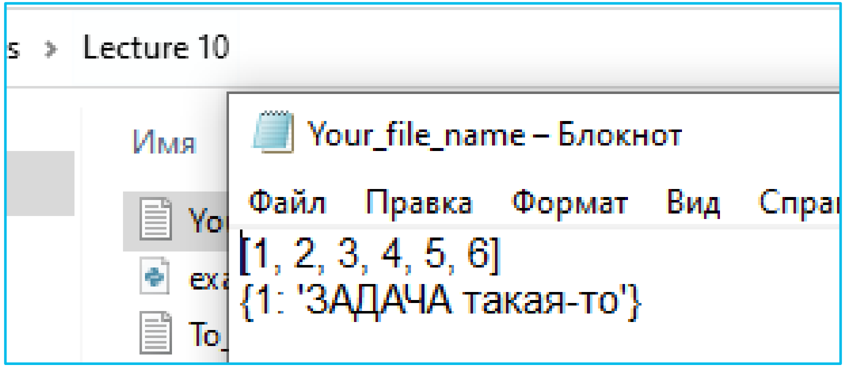
How to save without of using the absolute path?



```
c:\Users\stes7\Desktop\Py_projects\Lecture 10
Your_file_name.txt
c:\Users\stes7\Desktop\Py_projects\Lecture 10\Your_file_name.txt
```



Имя	Дата изменения	Тип	Размер
Your_file_name	15.06.2022 17:37	Текстовый докум...	1 КБ
example_path_to	15.06.2022 17:37	Исходный файл ...	1 КБ



```
[1, 2, 3, 4, 5, 6]
{1: 'ЗАДАЧА такая-то'}
```

https://stackoverflow.com/questions/25389095/python-get-path-of-root-project-structure

ЗАДАНИЯ

1) Прорешать всю классную работу
2) Выполнить все домашние задания

Почитать:

1) Byte of Python -
стр. 123-124,130-131, 132-135, 138-139

Крайний срок сдачи 19/10 в 21:00 (можно раньше, но не позже)

ЗАДАНИЯ

Название файлов, которые вы отправляете мне в telegram:

Vasia_Pupkin_class_work_L11.py,
Vasia_Pupkin_L11_1.py, Vasia_Pupkin_L11_1_1.py*, Vasia_Pupkin_L11_2.py,**
Vasia_Pupkin_L11_3_pizzeria.py, Vasia_Pupkin_L11_4_todo.py

То что со звездочками - желательно.

Vasia_Pupkin_L11_4_todo.zip/rar

Формат сообщения которое вы присылаете мне

(после полного выполнения домашнего задания, только один раз) в Telegram:

Добрый день/вечер. Я Вася Пупкин, и это мои домашние задания к лекции 11.

И присылаете файлы+архив с todo листом

Крайний срок сдачи 19/10 в 21:00 (можно раньше, но не позже)

https://docs.github.com/articles/using-pull-requests

Q&A

Create your possibilities. Bye bye.