

Lesson Overview

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [LESSON OVERVIEW](#)

Welcome in lesson 8! In this lesson we will learn about:

- **string formating** - until now, we have connected strings by pluses. It has worked, but it has not been so nice. Therefore we will start putting strings into strings. Goodbye for good, pluses!
- **text files** - how to write and read text files

Reminder: We are ending second month of this course, which means another bigger project for you. You should finish all projects, if you are thinking about getting job at IT company after course.

90% z Lekce 9

REVIEW EXERCISES

Leap Year

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [REVIEW EXERCISES](#) / [LEAP YEAR](#)

Create a function that will take in integer input representing year and will return **True**, if the year is leap, **False** otherwise. **Leap year occurs:**

- every four years,
- but every 100 years we skip a leap year,
- unless the year number is divisible by 400.

90% z Lekce 9

Example of leap years:

- years 1600, 2000 are leap, because they are divisible by 4 and even though they are divisible by 100 (what would discard them as leap years), they are divisible also by 400 and therefore they are leap
- years 1500, 1900 are divisible by 4, but they are also divisible by 100 and not divisible by 400 - therefore they are not leap
- years 1996, 2004 are leap because they are divisible by 4 and not divisible by 100 (so no need to be divisible by 400).

Example of function in use:

```
>>> is_leap(1600)
True
>>> is_leap(1700)
False
```

Online Python Editor

1 |

90% z Lekce 9

[spustit kód](#)

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

[Click to see our solution](#)

Number of days

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [REVIEW EXERCISES](#) / [NUMBER OF DAYS](#)

Create a function that will calculate the number of days between two dates. The dates inputs should be inserted as tuples of (year,month,day).

Do not forget that some years are leap therefore it is not possible to multiply the difference in years by 365.

You will need therefore to have helper function, that will tell you, whether a year is leap.

The other helper function could be the one, that increases the date by 1 day.

The first input into the main function should be date that comes earlier and the second should be the later one. If this is not the case, your program should swap the value between the input variables.

Example of function in use:

90% z Lekce 9

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

We will present two solutions:

1. Solution that increases counter of number of days by 1 each time the first date is increased towards the second date

Click to see our solution



2. Solution that counts numbers of days in individual years and adds them up

Click to see our solution



90% z Lekce 9

ONSITE PROJECT

Playing with formatting

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [ONSITE PROJECT](#) / [PLAYING WITH FORMATTING](#)

Our goal will be to create a program that will print given data nicely as a table:

Title1	Title 2	Title 3
abc	abc	abc
abc	abc	abc
abc	abc	abc
abc	abc	abc
abc	abc	abc
abc	abc	abc

Inserting string into a string

1. Formatting expression

```
"....%s....%d..." % (inserted_values)
```

Try it:

```
1 city = input('What city do you live in? ')
2 size = int(input('How many people live in your city? '))
3 how = 'big' if size > 1000 else 'small'
4 print('If there are %d people living in %s, the it is %s city' %
      (size,city,how))
```

2. Format() method

```
"...{}...{}..".format(*input_arguments)
```

90% z Lekce 9

```
1 celsius = int(input('What is the current temperature: '))  
2 fahrenheit = celsius * (9/5) + 32  
3 print('{} degrees Celsius would be {} degrees  
  Fahrenheit.'.format(celsius, fahrenheit))
```

Change the order of insertion

If needed, we can change the order in which the values are passed to the formatted string:

```
1 name = input('Your First Name: ')  
2 surname = input('Your Surname: ')  
3 greeting = 'Nice to meet you {1}, {0}.'  
4 print(greeting.format(name, surname))
```

Specify data insertion by key

This is very useful for populating templates like contracts, html templates etc. We will use this later, when working with files. Now just a demonstration:

```
1 n = input('Your First Name: ')  
2 s = input('Your Surname: ')  
3 greeting = 'Nice to meet you {surname}, {name}.'  
4 print(greeting.format(name=n, surname=s))
```

Unpacking into the format() method

There are two ways of unpacking:

1. single starred: ***arguments**
2. double starred: ****arguments**

Let's see how to use **single star** unpacking into **format()** method. Using single star unpacking we can unpack

1. sequences,

90% z Lekce 9

```
1 full_name = ('John', 'Smith')
2 greeting = 'Nice to meet you {1}, {0}.'
3 print(greeting.format(*full_name))
```

In order to perform **double starred** unpacking, we need to have a **dictionary**:

```
1 employee_data = {'name': 'John', 'surname': 'Smith', 'salary':1200,
2                  'job': 'Manager'}
3 template = '| {surname} | {name} | {salary} | {job} |'
4 print(template.format(**employee_data))
```

Format Specifiers

Format specifiers tell Python, how to depict input values. Format specifiers are separated from argument identifier by colon `:`. Argument identifier can be a keyname (format method's parameter name) or argument index or we can leave it empty.

The order of all the formatting specifiers is important

`{[keyname_or_index]:[[fill] align] [sign] [width] [,] [.precision] [typecode]}`

Let's train use of format specifiers on a table. We will populate it with the following data set:

```
1 data = [['Company', 'Date', 'Amount'],
2         ['Food Store', '2017-08-01', -112.56],
3         ['Gasoline', '2017-08-01', -655.605],
4         ['Energy', '2017-08-07', -1001.293],
5         ['Shoes&CO', '2017-08-09', -2130.321],
6         ['Electro', '2017-08-15', -1566.932],
7         ['Mother', '2017-08-22', 5000.00]]
```

Field Width

90% z Lekce 9

```
row_template = '| {:8} | {:8} | {:8} |'
```

Code Task

Print each row of the table providing each column with the width of 8 characters.

Value Alignment

We would like company names to be adjusted to the left, date to be centered inside its column and numeric values in the amount column to be right-justified.

To align values, we can use one of the following flags:

1. < - LEFT
2. > - RIGHT
3. ^ - CENTER

```
row_template = '| {:<8} | {:^8} | {:>8} |'
```

Code Task

Print each row of the table providing each column with the width of 8 characters while maintaining the above described alignment requirements.

Adjusting Precision

Maybe we want our numbers to be all aligned vertically as much as possible. In that case we need the precision flag, which is specified using dot and integer. All the numbers will follow the same **rounding** - precision rules:

```
row_template = '| {:<8} | {:^8} | {:>8.6} |'
```

Precision specifier can be used only on float values, not integers.

90% z Lekce 9

column to the left, centering the second column and making all the values in the last column right justified. All numbers in the last column can contain max 6 digits. Strings in the first column should be limited to max 8 characters.

Also, if we wanted to provide only a given number of characters to string fields, we can do that using the precision flag:

```
row_template = '| {:<8.8} | {:^8} | {:>8.6} |'
```

Filling empty spaces

Somebody decided, that it would be nicer, if empty spaces around dates in our table would be filled with underscores `_`. To do that, we need to supply fill format specifier, which should always follow the colon as the first specifier. The **Date** column should be made 12 characters wide:

```
row_template = '| {:<8.8} | {:_^12} | {:>8.6} |'
```

Fill parameter can be specified only if alignment parameter is provided as well.

Code Task

Let's implement the above requirements into our printing mechanism.

Forcing signs and separating thousands by commas

Sometimes, it is required to have numbers prepended with `+` or `-` sign. Sign forcing parameter goes between the alignment and width specifiers. To force depiction of both plus and minus signs we have to pass `+` symbol there. We will force the values in the last column to show both plus and minus signs:

```
row_template = '| {:<8.8} | {:_^12} | {:>+8.6} |'
```

Also, it is nicer, when thousands are separated by commas. To force this behaviour, we pass comma `,` between width and precision specifiers:

90% z Lekce 9

Parametrized format specifiers

Advanced use of formatting specifiers allows us to dynamically acquire values for format specifiers and then pass them into the template. We usually first calculate the minimum width of a column and then supply the calculated width values as inputs into the `format()` method:

Example of usage:

```
1 row_template = '| {:<{width1}.8} | {:_^{width1}} | {:>+{width1},.6} |'
2 min_widths = column_widths(table)
3 values = table[row_num]
4 row_template.format(*values, **min_widths)
```

Code Task

1. Create a function, that will calculate the minimum column widths for all columns in the data set.
2. Create a function that will format the column widths based on the calculated results from the function above.

Playing with Files

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [ONSITE PROJECT](#) / [PLAYING WITH FILES](#)

Create a new file called `test.txt` using text editor. The content of the newly created file should be:

```
1 This is the line number: 1
2 This is the line number: 2
```

Opening the File with Python

90% z Lekce 9

Function `open()` expects at least a string input representing the path to the file. This path can be relative or absolute. In the following example we use relative path:

```
1 file = open('test.txt')
```

Code Task

Open your file `test.txt`.

Reading the File

To read the file, we use the file object's `read()` method:

```
1 file.read()
```

Code Task

Read the content of the file you have just opened. And store it into the variable `content`.

Closing the File

To close the file, we use the file object's `close()` method:

```
1 file.close()
```

Code Task

Close the file.

Writing into the File

To write into the file, we need to specify the second parameter in the `open()` function:

90% z Lekce 9

we use `trag w` for write.

We cannot read from the file, if we open it under the `w` mode

Once we have our file open, we can write into it using its `write()` method. Write method should be given the content, we want to write into our file:

```
1 file.write('This is the line number: 3')
```

Code Task

In order we can work with our file, we need to open it again. Then we should write the following text into it - keeping the both on separate lines:

```
This is the line number: 3  
This is the line number: 4
```

Reading & Writing

Till now, we had to always close and re-open our file, if we wanted to read what was written or write. To avoid having to do this, we can open our file in `w+` or `r+` mode.

- `w+` - wipes the whole content of the file when opening - we begin to write into an empty file
- `r+` - keeps the original content

Code Task

Try to open your file using `r+` mode first. Read it and then write into it:

```
1 This is the line number: 5
```

Then close it and open it again in `w+` mode. Read it and then write into it:

```
1 This is the line number: 6
```

Navigating the File

To navigate the file, we can use file object's method `seek()`. We supply it with integer number to tell it, to which position we want to move. The first position in the file has index 0:

```
1 file.seek(0)
```

To find out, at what position the cursor encounters itself, we can use the `tell()` method:

```
1 file.tell()
```

Code Task

Close any open files you have in your terminal and open a new one in mode `r+`. Write the following strings into the file and then read them again:

```
1 This is the line number: 7
2 This is the line number: 8
```

Appending

Sometimes, especially when recording the activity of a program, we need to be able to append the information at the end of the file without overwriting what already existed inside the file. This is the case of so called **logging** where we could collect information about why our application has failed.

The `open()` function allows us to open the file in **append** mode using `a` flag only for writing and `a+` flag for reading and writing.

Code Task

- Open the file test.txt in mode `a+` and read it.
- Then move the cursor to the position (index) 0.

90% z Lekce 9

```
2 This is the line number: 10
```

- Move the cursor to the beginning of the file and read it again. The original content should be preserved.

Hangman with Memory

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [ONSITE PROJECT](#) / [HANGMAN WITH MEMORY](#)

Knowing how to store information in files, you can now use this knowledge to store the scores for the Hangman game you have implemented in the previous lesson as well as keep the database of words the program can choose from.

TASK 1

In your Hangman program, implement the functionality of retrieving random word from a file

TASK 2

In your Hangman program, implement the functionality of storing the results of the players in a file

STRING FORMATTING

What is String Formatting

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING](#) / [WHAT IS STRING FORMATTING](#)

String formatting is an operation that ought to make our strings:

- nicer
- easier to manipulate by inserting strings into strings

Let's say we would like to print to our terminal window **neatly organized table** like the one below:

Loan: 2000000 Years: 30 Interest: 693880 Monthly Payment: 7483					
=====					
Payment		Interest	Principal	Left to Pay	
=====					
1		4,692	2,791	2,693,880	
2		4,679	2,804	2,686,397	
3		4,666	2,817	2,678,914	
4		4,653	2,830	2,671,431	
5		4,640	2,843	2,663,948	
6		4,627	2,856	2,656,465	

Or we would like to avoid using **+** operator to concatenate strings:

```
>>> name = 'Bob'
```


90% z Lekce 9

```
my name is Bob and I am 37 years old.
```

We could just **use string formatting to insert the data**:

```
>>> name = 'Bob'
>>> age = 37
>>> print('My name is %s and I am %d years old.' %(name, age))
My name is Bob and I am 37 years old.
```

With all of this, string formatting can help us. There are two ways, how we can format strings in Python.

Two ways how to format

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING](#) / [TWO WAYS HOW TO FORMAT](#)

Throughout the next few slides, we will demonstrate string formatting using two methods:

1. formatting expression - `%()`
2. format method - `.format()`

Formatting expression

Expression uses an operator. In this case, we will use percent as formatting operator `%`. The operator is accompanied by the formatting flag `s` inside the string:

```
>>> name = 'Bob'
>>> formatted_string = 'My name is %s' %(name,)
>>> print(formatted_string)
My name is Bob
```

Don't worry, we will learn more about flags in the next section.

Format method

90% z Lekce 9

```
>>> name = 'Bob'
>>> formatted_string = 'My name is {}'.format(name)
>>> print(formatted_string)
My name is Bob
```

Which way to prefer?

Formatting expression is older approach to formatting and it does not offer so much functionality that the `format()` method provides. On the other hand, to some people, it may seem easier to use formatting expression. The choice is yours.

Insertion

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING](#) / [INSERTION](#)

We have already briefly shown you, how the formatting expression and method insert data into a string. Now we will add some more detail to it.

Formatting Expression

Formatting expression consists of three parts:

1. formatted string containing **conversion specifiers**,
2. percent sign separating formatted string from a tuple of values to be inserted
3. and a tuple of values to be inserted.

```
"....%s....%d..." % (inserted_values)
```

There are many conversion specifiers, but at the moment we can get by with two conversion specifiers:

- **s** - tells Python, that the inserted item should be formatted as a **string**

90% z Lekce 9

Placeholders are matched with the values in the tuple based on the order. The first value in the tuple is passed to the place of the first formatting specifier, second to the second place, etc.

```
>>> name = 'Bob'
>>> age = 37
>>> print('My name is %s and I am %d years old.' %(name, age))
My name is Bob and I am 37 years old.
```

Format Method

In the case for format method, the formatted string should contain placeholders written using curly braces `{}`. Arguments passed to `format()` method are then assigned to each placeholder based on their order in the method's call parentheses.

The method's syntax: `"...{}...{}...".format(*input_arguments)`

```
>>> name = 'Bob'
>>> age = 37
>>> formatted = 'My name is {} and I am {} years old.'.format(name, age)
My name is Bob and I am 37 years old.
```

Data type conversion

It is important to point out, that in both cases (expression & method), you do not need to change numeric data types into strings as it was with string concatenation using the plus (+) sign:

```
>>> 'Today is our {}st anniversary.'.format(1)
Today is our 1st anniversary.
>>> 'Today is our %dst anniversary.' %(1)
Today is our 1st anniversary.
```

Changing insertion order

90% z Lekce 9

One of the advantages of `.format()` method is that it allows us to change the order in which the arguments values are passed to placeholders inside the string. All we need to do is to specify the index number inside the curly braces:

```
'{1} and {0}'.format(arg1, arg2)
```

Example:

```
>>> name = 'Bob'
>>> age = 37
>>> formatted = 'My name is {1} and I am {0} years old.'.format(name,
age)
>>> formatted
'My name is 37 and I am Bob years old.'
```

Of course, if we try to pass **numbers higher than the highest index** of the last argument inside the parentheses, we get an **error**:

```
>>> formatted = 'My name is {2} and I am {1} years old.'.format(name,
age)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

Passing index number allows us to **repeat the same value** multiple times:

```
>>> formatted = 'My name is {0} and I am {0} years old.'.format(name)
>>> formatted
'My name is Bob and I am Bob years old.'
```

In the example above we need to pass numbers inside the curly braces, otherwise, Python will expect two arguments in the `format()` method:

```
>>> formatted = 'My name is {} and I am {} years old.'.format(name)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

90% z Lekce 9

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING](#) / [INSERTION BY KEY](#)

Besides matching the placeholders and inputs by their order, we can also use specification by key. Matching placeholders with input values by keys is supported by both formatting expression and `.format()` method.

Formatting Expression

The first thing that passing by key should remind us is the dictionary data type. Instead of using tuple following the `%` operator, we can supply a dictionary object.

Inside the formatted string we need to specify the key name enclosed in parentheses, telling Python, which value we want to be supplied to that spot. The key name goes **between** the `%` and the formatting specifier.

Keys specified inside formatted string should match the keys inside the dictionary. Dictionary keys should be also of type string.

```
person_data = {'name': 'Bob', 'age' : 37}
>>> formatted = 'My name is %(name)s and I am %(age)d years old.' %
person_data
>>> formatted
'My name is Bob and I am 37 years old.'
```

Format Method

```
>>> formatted = 'My name is {name} and I am {age} years
old.'.format(name='Bob', age=37)
>>> formatted
'My name is Bob and I am 37 years old.'
```

Note that the keywords within the format method are parameters - they are not strings, unlike f.e. the keys in a dictionary.

Now what if we had our values already stored inside global variables, we can pass these variables to the `format()` method:

90% z Lekce 9

```
>>> formatted = 'My name is {name} and I am {age} years  
old.'.format(name=name, age=age)  
>>> formatted  
'My name is Bob and I am 37 years old.'
```

Here we can nicely see the difference among function parameters and arguments. Arguments are global variables **name** and **age** and parameters **name** and **age** are local variables.

Unpacking to format method

[PYTHON ACADE...](#) / [8. STRING FORMATTING & TEXT FIL...](#) / [STRING FORMATTI...](#) / [UNPACKING TO FORMAT METH...](#)

We can unpack with the help of two different operators: ****** and *****.

We use **two stars for dictionaries**. It is very convenient use of passing values by keys into the formatted method is its combination with argument unpacking in form of dictionary. We have already seen this with formatting expression. In case of **format()** method we just unpack the dictionary inside the function call parentheses:

```
>>> person_data = {'name': 'Bob', 'age' : 37}  
>>> formatted = 'My name is {name} and I am {age} years  
old.'.format(**person_data)  
>>> formatted  
'My name is Bob and I am 37 years old.'
```

We use **one star for sequences**. In this case, unpacked argument values will be matched with placeholders by their position in the unpacked sequence:

```
>>> data = ('Bob', 37)  
>>> formatted = 'My name is {} and I am {} years old.'.format(*data)  
>>> formatted  
'My name is Bob and I am 37 years old.'
```

90% z Lekce 9

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING](#) / [VALUE FORMATTING](#)

We already know, how to pass values into strings. Now we will look at the techniques, how those values can be formatted - that means, how can we determine the way the value will look like inside the string.

What we may want to influence is:

1. Field Width
2. Value Padding and Alignment
3. Number Rounding
4. ...

We will mostly focus on the value formatting with the format method.

Format method specifiers

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING](#) / [FORMAT METHOD SPECIFIERS](#)

Formatting can be configured using many formatting specifiers. These have to be listed in defined order. Below, we have listed them all, in order you get an idea. We first and mainly focus on the specifiers of the format method.

Format specifications are listed inside curly braces. We need to divide the key name or index specifier from the rest by a colon `:`.

```
'This is the formatted text: {[keyname_or_index]:[[fill] align] [sign]  
[width] [,] [.precision] [typecode}]'.format(keyname)
```

- **keyname_or_index** - determine, which argument should replace this placeholder
- **fill** - Can be any character other than "{" or "}". Here we provide the character, that should be filled in instead of white space if field width is greater than item length. This option can be specified **only if align is specified**.

90% z Lekce 9

space

- **width** - Can be an integer value. Specifies the length of the field. If the length specified is less than input string length, then the width parameter is ignored.
- **,(comma)** - Possible value `,`. Requests comma to mark thousands (inserted after every 3 zeroes) in numbers
- **precision** - Possible value `.integer`. Specifies maximum input string length or if argument is float, then max number of decimal places.
- **typecode** - refers to flags we already know such as `f` (float) or `d` (decimal)

Width

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING](#) / [WIDTH](#)

We can tell how many characters a field, where we want to insert our string should have, by passing in an integer value after the colon:

```
>>> '|{0:3}|{0:4}|'.format(45)
'| 45| 45|'
```

Integer value is automatically converted into a string. The input number above consists of 2 digits. The first placeholder tells Python to take the argument at index 0 and insert it into a field 3 characters wide and the second does the same for field 4 characters wide. We use the "|" character to demonstrate that how inserted values are incorporated into the string.

We can see, that field 3 characters long fills the empty space by space character " " .

Alignment and Fill

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING](#) / [ALIGNMENT AND FILL](#)

90% z Lekce 9

may want to fill in the dollar signs instead of white spaces. Maybe something like this:

```
'|45$|45$$|'
```

To be able to use fill character, we need to specify value alignment as well:

- value is left aligned if we pass in `<` symbol
- value is right aligned if we pass in `>` symbol
- value is centered if we pass in `^` symbol

The order of formatting specifiers has to be: `{[key_or_index] : [fill] [alignment] [width]}` and it looks something like this:

```
>>> '|{0:$<3}|{0:$<4}|'.format(45)
'|45$|45$$|'
```

Precision

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING](#) / [PRECISION](#)

The precision parameter specifies how many numbers are permitted for floating point numbers. Precision is signaled using dot `.` and integer.

```
>>> '|{0:.5}|'.format(123.4567)
'|123.46|'
```

Precision `.5` tells python, that 5 digits are allowed. In this case the number had to be rounded to two decimal values as the total number of digits is be 5.

We can play with the precision value to see, how the number inside the string is depicted. Once the precision specifier falls below the number of non-decimal digits, exponential notation (`e+02`) is used.

```
>>> '|{0:.3}|{0:.4}|{0:.5}|'.format(123.4567)
```

90% z Lekce 9

```
>>> '|{0:$<6.5}|'.format(1.234)
'|1.234$| '
>>> '|{0:$<7.5}|'.format(1.234)
'|1.234$$| '
```

Precision can be used also on string arguments. In that case only as many characters are depicted as given by the precision.

```
>>> '|{0:.4}|'.format('Hello')
'|Hell| '
```

Comma

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING](#) / [COMMA](#)

A nice feature that improves number readability is to include comma to separate thousands.

```
>>> '|{0:,.}|'.format(123456789)
'|123,456,789| '
```

Combined with other featurers, it looks like this:

```
>>> '|{0:*>12,}|'.format(123456789)
'|*123,456,789| '
```

And including precision:

```
>>> '|{0:*>19,.14}|'.format(123456789.123456)
'|**123,456,789.12346| '
```

The precision value will include commas as well, meanwhile the precision tells us, how many digits are permitted in one field.

90% z Lekce 9

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING](#) / [FORCING SIGNS](#)

Following the align flag, we can put the sign forcing flag. We have 3 possible flags:

- **-** - indicates that a sign should be used only for negative numbers (this is the default behavior).
- **+** - indicates that a sign should be used for both positive as well as negative numbers
- **space** - indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers

```
>>> '|{0:-}|'.format(280)
'|280|'
>>> '|{0: +}|'.format(280)
'|+280|'
>>> '|{0: }|'.format(280)
'| 280|'
```

Other Formatting Methods

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING](#) / [OTHER FORMATTING METHODS](#)

Besides the **.format()** method, there are other, very focused methods, that help with string formatting:

Syntax	Description	Example	Output
--------	-------------	---------	--------

90% z Lekce 9

field width and optionally allows us to provide character, that will be filled to the **right** of the string, if the field width is greater than string length.

`.ljust(num, 'char')``"5".ljust(5, "x")``'5xxxx'`

Lets us to specify the field width and optionally allows us to provide character, that will be filled to the **left** of the string, if the field width is greater than string length.

`.rjust(num, 'char')``"5".rjust(5, "x")``'xxxx5'`

Lets us to specify the field width and optionally allows us to provide character, that will be filled **around** the string, if the field width is greater than string length.

`.center(num, 'char')``"5".center(5, "x")``'xx5xx'`

90% z Lekce 9

only the field width
because the filling
character is **"0"** that
will be filled to the left
of the string, if the
field width is greater
than string length.

`.zfill(num, 'char')` `"5".zfill(5, "x")` `'00005'`

All these methods are additions to the ones we already know from [lesson one](#). You can revise them if you'd like ;)

STRING FORMATTING +

90% z Lekce 9

[PYTHON ACAD... / 8. STRING FORMATTING & TEXT FILES / STRING FORMATTING... / FORMATTING EXPRESSIONS OF LEI...](#)

Here format specifiers are listed behind the sign `%`.

```
'This is the formatted text:  %[(keyname_or_index)][flags][width]
[.precision]conversion_type' % (keyname)
```

- **keyname** - we have already learned, that values can be passed into formatted strings using keys
- **flags** - specifications of alignment, fill character
 - `#` mostly used with `x` and `o` conversion type to prepend it with `0o` or `0x` prefix
 - `0` The conversion will be zero padded for numeric values.
 - `-` The converted value is left adjusted (overrides the `0` conversion if both are given).
 - `' '` (a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
 - `+` A sign character (`+` or `-`) will precede the conversion (overrides a space - `' '` flag).
- **width** - tells Python, how many characters are permitted for a given placeholder
- **precision** - specifies the number of decimal places of a float value
- **conversion_type** - we have already learned about `s` and `d` conversion type.

We provide few examples below. Some use cases are analogous to the `format()` method.

Width

```
>>> '|%3d|' % (54,)
'| 54|'
```

Fill

Here we can use only digit `0` to fill in the spaces, that are left empty:

90% z Lekce 9

Alignment

Inside formatting expressions, we use `-` sign to align the input value to the left. Overrides the '0' character fill

```
>>> '|%-03d|' % (54,)
'|54|'
>>> '|%-3d|' % (54,)
'|54|'
```

Precision

We need to use `f` conversion type instead of `d`

```
>>> '|%3.3d|' % (54.653,)
'|054|'
>>> '|%3.3f|' % (54.653,)
'|54.653|'
```

Forcing Sign

We use `+` sign to show number valence:

```
>>> '|%+3.3f|' % (54.653,)
'|+54.653|'
>>> '|%+3.3f|' % (-54.653,)
'| -54.653|'
```

Formatting Expression Dynamic width

[PYTHON ACAD...](#) / [8. STRING FORMATTING & TEXT ...](#) / [STRING FORMATTI...](#) / [FORMATTING EXPRESSION DYNAMIC ...](#)

90% z Lekce 9

```

' |      45 | 000045 | 45      |      +45 |      45 | '
>>> ' |%#6s|%06s|%-6s|+%6s|% 6s| ' % ('Bob', 'Bob', 'Bob', 'Bob', 'Bob')
' |      Bob |      Bob | Bob      |      Bob |      Bob | '

```

If we wanted to calculate the field width first, we are forced into the rather cumbersome syntax. We can specify dynamic width using asterisk (*), the actual width is read from the preceding element in the tuple of input values:

```

>>> num1 = 45/3; width1 = len(str(num1)) + 2
>>> num2 = 45*3; width2 = len(str(num2)) + 2
>>> num3 = 45%3; width3 = len(str(num3)) + 2
>>> ' |%#*d|%0*d|%-*d|+*d|% *d| ' % (width1,num1, width1,num2,width2,
num3,width3, 45678,width2, 45)
' |      15 | 000135 | 0      | +45678 |      45 | '

```

When using asterisk, the number of items in the input tuple has to be doubled. That way, taken in order, every odd item determines the field width and every other item determines the actual input value.

Formatting Expression Precision

PYTHON ACADE... / 8. STRING FORMATTING & TEXT FI... / STRING FORMATTIN... / FORMATTING EXPRESSION PRECI...

Given as a . (dot) followed by the precision value.

When determining precision, we have to have in mind that we need to convert the input values into float type. For that purpose we need conversion type flags:

- f - classic floating point notation
- e - provides scientific notation with exponent "e"
- g - uses exponential format "e" if the exponent is less than -4 or not less than precision, and decimal format f otherwise, with a default total digits precision of 6

```

>>> num = 1 1234567

```


If specified as `*` (an asterisk), the actual width is read from the next element of the tuple in values, and the value to convert comes after the precision.

```
>>> num1 = 45/3; prec1 = len(str(num1)) + 2
>>> num2 = 45*3; prec2 = len(str(num2)) + 2
>>> num3 = 45%3; prec3 = len(str(num3)) + 2
>>> '|#6.*d|%06.*d|%-.*d| %+6.*d| % 6.*d|' % (prec1,num1,
prec1,num2,prec2, num3,prec3, 45678,prec1, 45)
'|000015|000135|00000|+45678| 000045|'
```

Escaping Curly Braces

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING +](#) / [ESCAPING CURLY BRACES](#)

We can escape curly braces and though achieve that they will be printed inside the string by duplicating them:

```
>>> print('This is not a placeholder {{0}}.'.format("input"))
This is not a placeholder {0}.
```

If we wanted to make a placeholder that would print not only the input value but surround it with curly braces as well, we need to have 3 braces on each side (two for escaping, one for creating placeholder):

```
>>> print('This is a placeholder {{{0}}}.'.format("input"))
This is a placeholder {input}.
```

Parametrized Formats

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING +](#) / [PARAMETRIZED FORMATS](#)

90% z Lekce 9

```
>>> '|{:align}{width}.{precision}f|'.format(123.4567, align='>',  
width=8,precision=5)  
'|123.45670|'
```

We have combined positional and keyword arguments. Keyword arguments specify values for formatting options. Positional argument is the actual value that is being formatted.

Applying placeholders to formatting options provides us with great flexibility in formatting.

Other Conversion Flags

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [STRING FORMATTING +](#) / [OTHER CONVERSION FLAGS](#)

The last flag among format specifiers is the type code. It determines how the data should be presented. There is a wide range of type flags, therefore we will divide them into 3 groups and then we will demonstrate the workings of only few of them:

The 3 type groups, we distinguish are:

1. **String presentation types** - **s** - does not have to be filled in as it is the **default option**

```
>>> '|{0:*<6s}|'.format('Hello')  
'|Hello*|'
```

Be careful, we cannot use this specifier with numeric data type inputs

```
>>> '|{0:*<6s}|'.format(12345)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: Unknown format code 's' for object of type 'int'
```

2. Integer presentation types

Integer presentation flags are mostly related to the conversion among numeric representations (decimal - binary - hexadecimal - octal)

90% z Lekce 9

```
>>> '|{0:*<6b}|'.format(280)
'|100011000|'
>>> '|{0:*<6o}|'.format(280)
'|430***|'
>>> '|{0:*<6x}|'.format(280)
'|118***|'
```

It is also possible to convert number into corresponding character (computers encode characters into numbers):

```
>>> '|{0:*<6c}|'.format(280)
'|Ě*****|'
```

3. Presentation types for floating point and decimal values

In case we would like to generate scientific reports, we may make use of the following formatting options. We will use flags **e**, **f** to present numbers in exponential notation:

- **e** - Prints the number in scientific notation using the letter 'e' to indicate the exponent. The default precision is 6

```
>>> '|{0:e}|'.format(12345)
'|1.234500e+04|'
```

- **f** - Displays the number as a fixed-point number. The default precision is 6.

```
>>> '|{0:f}|'.format(12345)
'|12345.000000|'
>>> '|{0:f}|'.format(123.456754646464)
'|123.456755|'
```

TEXT FILES

What a file is?

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / [WHAT A FILE IS?](#)

Files are in Python program again represented as objects. The best known types of files that an average user of computer knows, is text file. We know that we can do the following operations with text files:

1. open - **create an object representation of a file**, that resides on a hard drive for example
2. read - **load the contents of the file**
3. write - **record new data to the file**
4. close - **save and close a file**

It is good to know, that text files are not the only representatives of file-like objects. We can have also so called **binary files**, which are for example image files.

In the following few sections we will learn how to work with text files in Python program.

File objects are sometimes called **streams**.

90% z Lekce 9

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / FILE PATH

Before we can open a file in our Python program, we need to know, where it resides. Also, we need to tell Python somehow, we want it to go to that place in memory in order to load the file.

To tell Python, which file we want to load, we specify so called **file path**. An example of a file path could be:

- **Windows:** `C:\Users\Bob\Documents\gifts.txt`
- **Linux and Mac:** `/home/Bob/Documents/gifts.txt`

We can see, that file paths are written with backslash `'\'` in Windows and forward slash `'/'` in Linux or Mac OS (let's call them both Unix) separating folders. They specify the starting point (the leftmost name) and the end point (the rightmost name). Starting point usually refers to a folder name and end point designates the file, we want to load.

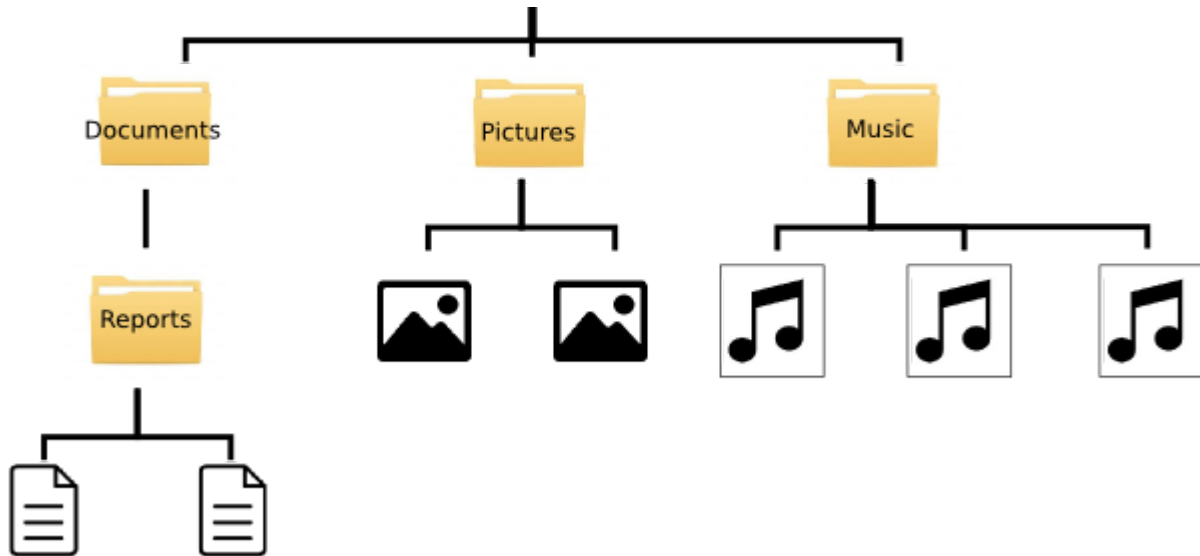
The above file paths specify the location of file `gifts.txt` that is placed inside the **Documents** folder. Documents folder is located inside the folder called **Bob** etc.

But file path is also considered a single filename: `gifts.txt`. In that case the, file is considered to be placed in the **current working directory - CWD** (more about CWD in the next section).

File path is therefore a specification of a **unique location of a file in a file system**, listing all the directories leading to the given file.

Files live in directories and directories can be nested in other directories, thus creating a **hierarchical tree** of files and folders.

90% z Lekce 9



Absolute vs. Relative Path

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / [ABSOLUTE VS. RELATIVE PATH](#)

In the previous section, we have shown you the following file paths:

- **Windows:** `C:\Users\Bob\Documents\gifts.txt`
- **Linux and Mac:** `/home/Bob/Documents/gifts.txt`

Absolute path

Both these paths are so called absolute paths. In Windows, absolute paths begin with drive label (`'C:'`), in Unix with forward slash `'/'` . We call this starting points root. Absolute file paths start always at root file.

Relative path

A relative file path is interpreted from the perspective of (starts at) the current working directory.

Current working directory is the directory in which a user is working at a given time or directory in which a program is executed.

90% z Lekce 9

another folder called `Images`. This folder could contain images manipulated by our Python script. In order the script can access a image `cat.jpg`, we can give it a relative path to access it:

`Images/cat.jpg`

This path does not contain the root directory and therefore Python will begin the search for the file from the current working directory `Scripts` of the Python script `invert_image.py`. It will first look, whether the `Scripts` folder contains another folder called `Images`. When found, then it will continue searching for the file named `cat.jpg`.

We have prepared for you a task **Simulate path search** in the section Exercises, where by solving the task, you can better understand, what a computer has to do in order to find files in the file system.

Opening a file

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / [OPENING A FILE](#)

In order we can read or write into our files, we first need to create them in our program.

File is represented as a **file object** in Python program. We can create file object by using built-in function `open()`. We can open a text file that resides at the file path `/home/Bob/Documents/gifts.txt` as follows:

```
>>> file = open('/home/Bob/Documents/gifts.txt')
```

Note that the **path to the target file is a string**.

You can also check [Python documentation](#) concerning function `open()`. We will learn more about `open()` in the upcoming sections.

Code Task

90% z Lekce 9

```
1 This is the first line
2 This is the second line
```

2. Save and close the file.

3. Open the terminal window and change your current working directory to **PythonBeginner/Lesson6**.

4. Launch Python interpreter (command **python** or **python3**) in the terminal window and write the following code:

```
>>> file = open('test.txt')
```

We are using relative path to locate and then open the text file.

5. Now pass the variable **file** into the print function - observe, what gets printed

```
>>> print(file)
<_io.TextIOWrapper name='test.txt' mode='r' encoding='UTF-8'>
```

We get back the **file object** variable **file** is representing.

Opening a non-existent file

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / [OPENING A NON-EXISTENT FILE](#)

If the file, we are trying to open does not exist, we get **FileNotFoundError**:

```
>>> file = open('nonexistent_file.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'nonexistent_file.txt'
```


90% z Lekce 9

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / [READING A FILE](#)

The main purpose of files is that they can be written information in and read the information out. We have already created a file called `test.txt` and populated it with some text. Now we want to learn, how we can read the text from the file object in our programs.

There are multiple methods, that allow us to read from file object. Here is the most commonly used `read()` method, that returns the string, that represents the text the file contains.

We have our file open:

```
>>> file = open('/home/Bob/Documents/test.txt')
```

Now we will read it:

```
>>> text = file.read()
```

What will happen, if we write the variable name `text` into the terminal?

```
>>> text
'This is the first line\nThis is the second line'
```

All the characters, including the special character `'\n'` get printed to the terminal.

Now let's try to pass the variable into the `print()` function:

```
>>> print(text)
This is the first line
This is the second line
```

Using `print()` cause the text to be nicely formatted.

Code Task

If you have not followed the steps described above, please, try to perform them now. Only that way you will get the feel for opening files in Python.

90% z Lekce 9

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / [CLOSING A FILE](#)

Our file object was opened in so called **read-only mode**. We have already read the information the file object contained:

```
>>> text = file.read()
>>> text
'This is the first line\nThis is the second line'
>>> print(text)
This is the first line
This is the second line
```

If we now try to do read it again, we will not receive any more data:

```
>>> more_text = file.read()
>>> print(more_text)
>>>
```

Now we can close the file using the **close()** method:

```
>>> file.close()
```

Every file should be closed, once we are done working with it. If we do not do that and we will keep opening new file objects, our computer memory can get cluttered pretty soon.

If we try to read from a closed file, we get an error:

```
>>> file.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Writing into a file

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / [WRITING INTO A FILE](#)

90% z Lekce 9

in writing mode by supplying the second argument to the `open()` function - string `'w'`.

```
>>> file = open('test.txt', 'w')
```

If we write the `file` variable into the terminal, we get the following file object description:

```
>>> file
<_io.TextIOWrapper name='test.txt' mode='w' encoding='UTF-8'>
```

We can see that the file mode is equal to `'w'`. We can now use the `write()` method to write the strings we want to record in the file:

```
>>> file.write('This is the third line\nThis is the fourth line')
46
```

The `write()` method returns a number referring to the number of characters, that have been written into the file.

Now we probably want to see, what has been written into our file, by reading it back. But we have a problem:

```
>>> file.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
io.UnsupportedOperation: not readable
```

File object opened in `'w'` mode do not support reading operation. In order to read it, we need to close it and open it again in a mode, that will allow us to read and write at the same time. More on that in the next section.

```
>>> file.close()
```

Navigating Text Files

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / [NAVIGATING TEXT FILES](#)

90% z Lekce 9

provides two methods to navigate through the file content.

We can imagine navigation across the file as movement of cursor in text editor program.

1. Determine the current position

To find out the position, where the virtual cursor encounters itself, we can use the `tell()` method:

```
>>> file = open('test.txt')
>>> file.tell()
0
```

After opening the file in `'r'` and `'w'` mode, the cursor encounters itself at the first position. We can now read the file:

```
>>> file.read()
'This is the third line\nThis is the fourth line'
>>> file.tell()
47
```

After reading the file, the cursor moves to the end of the file. How can we re-read the file without closing it?

2. Change the cursor position

To move the cursor to any given position in the file, we need to use the `seek()` method. We continue working with the above opened file:

```
>>> file.seek(0)
0
>>> file.tell()
0
>>> file.read()
'This is the third line\nThis is the fourth line'
```

By giving the `seek()` method input in form of integer, we tell it to move the cursor to that index (position) in the file. Above, we have decided to move to the index 0 (the first position). That way

Reading and Writing

PYTHON ACADEMY / 8. STRING FORMATTING & TEXT FILES / TEXT FILES / READING AND WRITING

In scenarios, when we want to write into a file and then again read, what has been written so far, we need a better solution, that opening a file in write mode, then closing it and then opening it in read mode.

Fortunately, besides the `'w'` flag for write-only mode (and `'r'` for read-only, which is optional) Python recognizes the following file mode flags inside the `open()` function:

- `'r+'`
- `'w+'`
- `'a'`
- `'a+'`

Before we explore, how a file object behaves under each of the above flags, we should check, what our file currently contains:

```
>>> file = open('test.txt')
>>> text = file.read()
>>> print(text)
This is the third line
This is the fourth line
```

We can see that the original content of the file has been replaced by the string, we have written into the file in the previous section.

That means that the `'w'` mode causes Python to overwrite the whole content, that has originally been in the file.

Let's explore the above read-write options.

90% z Lekce 9

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / [USING W+ FLAG](#)

Files opened with the `'w+'` allow for reading and writing into the files. We should be careful about the fact, that whatever content was inside the file before, will be erased at the moment, when we open the file with `'w+'` mode:

Originally, the file contained the following string:

```
'This is the third line\nThis is the fourth line'
```

Now we will open the file using the flag `'w+'` :

```
>>> file = open('test.txt', 'w+')
>>> file.read()
''
```

We can see all the original content has been erased. Now we can write the new content:

```
>>> file.write('This is the fifth line\nThis is the sixth line')
45
>>> file.read()
''
>>> file.seek(0)
0
>>> file.read()
'This is the fifth line\nThis is the sixth line'
>>>
```

In order to read the whole content that has just been written, we need to navigate to the beginning of the file first. Then we can use the `read()` method.

Using r+ flag

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / [USING R+ FLAG](#)

90% z Lekce 9

file. That means that the original content will be gradually overwritten by the newly written content.

```
>>> file = open('test.txt', 'r+')
>>> file.tell()
0
>>> file.read()
'This is the fifth line\nThis is the sixth line'
>>> file.tell()
45
>>> file.write('\nThis is the seventh line')
25
>>> file.seek(0
... )
0
>>> file.read()
'This is the fifth line\nThis is the sixth line\nThis is the seventh
line'
```

In the code above, after opening the file, we first determine our current position in the file. Then we read the whole content, what moves our current position to the end of the file. Finally we write a new string at the end of the file and read the whole content again.

Using a and a+ flag

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / [USING A AND A+ FLAG](#)

In this case **'a'** is a shorthand for **append**. A file, that is opened in this mode can be appended new data always after the original content:

```
>>> file = open('test.txt', 'a')
>>> file.write('\nThis is the eighth line')
24
```

90% z Lekce 9

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
io.UnsupportedOperation: not readable
```

Reading as well as writing in **append** mode is supported by files opened with **'a+'** flag:

```
>>> file.close()  
>>> file = open('test.txt', 'a+')  
>>> file.read()  
''  
>>> file.seek(0)  
0  
>>> print(file.read())  
This is the fifth line  
This is the sixth line  
This is the seventh line  
This is the eighth line  
>>> file.write('\nThis is the ninth line')  
23  
>>> file.seek(0)  
0  
>>> print(file.read())  
This is the fifth line  
This is the sixth line  
This is the seventh line  
This is the eighth line  
This is the ninth line
```

File modes summary

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES](#) / [FILE MODES SUMMARY](#)

90% z Lekce 9

Read or write

Mode	Description
'r'	File can be only read but not written into
'w'	File can be only written into. The original content is overwritten
'a'	File can be only written into. The original content is kept, new content is appended at the end

Read and write

Mode	Description
'r+'	Original content is kept and we can write wherever we want inside the file
'w+'	Original content is erased
'a+'	Original content is kept and we can write only at the end of the file

TEXT FILES +

Looping over a file

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES +](#) / [LOOPING OVER A FILE](#)

We can use a for loop on file objects. Each loop cycle, Python reads the content of the following line and puts it into the loop variable. Lines are considered strings separated by the newline character `'\n'`:

```
>>> file = open('test.txt', 'w+')
>>> file.write('This is the first line\nThis is the second line\nThis is
the third line')
69
>>> file.seek(0)
0
>>> for line in file:
...     print(line, end='')
...
This is the first line
This is the second line
This is the third line
>>>
```

Don't forget to try the above listed commands in your Python interactive console ;)

Besides calling file object's `read()` method, there the following ways how a file can be read:

- `file.read(N)`

We can specify, how many characters we want to be read by passing a number `N` into the `read()` method:

```
>>> file.read()
'This is the third line\nThis is the fourth line'
>>> file.seek(0)
0
>>> file.read(10)
'This is th'
```

- `file.readline()`

This methods reads everything up to the newline character `'\n'` or the end of the file:

```
>>> file.read()
'This is the third line\nThis is the fourth line'
>>> file.seek(0)
0
>>> file.readline()
'This is the third line\n'
```

- `file.readlines()`

This method splits the string being read on newline character `'\n'`) and puts the individual lines of text into a list:

```
>>> file.seek(0)
0
>>> file.readlines()
['This is the third line\n', 'This is the fourth line']
```

90% z Lekce 9

```
>>> file.seek(0)
0
>>> file.readlines(1)
['This is the third line\n']
>>>
```

Other ways of writing into files

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES +](#) / [OTHER WAYS OF WRITING INTO FILES](#)

Besides using bare `file.write()` method, we can write into files using `writelines()` method. Method accepts a list of strings as an input. Strings are automatically concatenated before they are written into the file:

```
>>> file = open('test.txt', 'w')
>>> data = ['Hello', 'Mr', 'Bob']
>>> file.writelines(data)
>>> file.close()
>>> file = open('test.txt')
>>> print(file.read())
HelloMrBob
```

More about navigation

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES +](#) / [MORE ABOUT NAVIGATION](#)

Method `seek()` that belongs to file object can take more than one input. We have learned its usage with only one input, which specified the index to which we want to move the file cursor. The second argument specifies the relative position against which the offset (cursor movement) is performed.

90% z Lekce 9

Offset relative to the file beginning

Offset relative to the beginning is performed if the second argument is **set to 0** or it is not set at all (it is a default value for the second parameter).

```
>>> file = open('test.txt')
>>> file.read()
'This is the third line\nThis is the fourth line'
>>> file.seek(5)
>>> file.read()
'is the third line\nThis is the fourth line'
```

Offset relative to the file end

If we wanted to move cursor to the end of the file, without reading the file we can set the **second argument to value 2**. This operation is possible only if the first argument (offset) is equal to 0:

```
>>> file.seek(20)
20
>>> file.seek(0,2)
46
```

Binary files

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES +](#) / [BINARY FILES](#)

In this lesson we are learning basics of working with text files. However huge amount of files out there are so called binary files. The name 'binary files' may sound scary for those who have never heard it. But it doesn't have to. An example of binary file is simply **an image or video or music file**. All these files cannot be converted into plain text files.

These files can be opened only in binary mode. That means that we have to append **b** flag after file mode flag:

90% z Lekce 9

```
image = open('img_2011.jpg', 'rb')
```

- For writing

```
>>> image = open('img_2011.jpg', 'wb')
```

For now this is all we need to know about binary files.

File object attributes

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES +](#) / [FILE OBJECT ATTRIBUTES](#)

We can determine few facts about our file objects just by asking for its attribute values.

File name

We can find out file's name by asking for `.name` attribute that should be associated with the file object using dot `.`:

```
>>> file = open('test.txt')
>>> file.name
'test.txt'
```

File mode

Sometimes it could be useful to find out, how a file will behave, if we will begin to write into it or read it. To find this out, we need to determine its mode:

```
>>> file.mode
'r'
```

File size

90% z Lekce 9

```
>>> import sys
>>> sys.getsizeof(file)
216
```

Context Manager

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [TEXT FILES +](#) / [CONTEXT MANAGER](#)

In this section we will provide a brief introduction into the concept context manager.

Context management is done on objects that support it. One such kind of objects are file objects. For context manager to work, the object has to support open (enter) and close (exit) operations.

The main goal is to create a context in which the object is opened and then lives. Once all the prescribed commands, listed in the context are executed, the file is automatically closed. That means, that by using context manager, we do not have to be careful with closing a file explicitly (`file.close()`). This is done automatically, even if the program terminates in error, that could be raised inside the context manager.

Syntax:

```
with object_supporting_context_manager as alias:
    statements to be performed with the object in the context
```

An example of opening, writing into and closing file:

```
>>> with open('test.txt', 'a') as file:
...     file.write('This is the last line')
...
21
```

If we tried to perform read operation on the file object above, we would be informed, that the file is closed:

90% z Lekce 9

```
File <stdin>, line 1, in <module>  
ValueError: I/O operation on closed file.
```

String Formatting

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [QUIZ](#) / [STRING FORMATTING](#)

1/15

Which formatting expression is correct?

90% z Lekce 9



A. 'My name is %s' % ('Bob',)

B. 'My name is %s' % ('Bob',)

C. 'My name is \$s' \$ ('Bob',)

D. 'My name is @s' @ ('Bob',)

DALŠÍ OTÁZKA

Text Files

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [QUIZ](#) / [TEXT FILES](#)

1/12

Which of the following paths is a **absolute path** in Windows OS?

A. /User/Username/Documents/report.doc

B. User/Username/Documents/report.doc

C. C:\User\Username\Documents\report.doc

90% z Lekce 9

Column Widths

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [HOME EXERCISES](#) / [COLUMN WIDTHS](#)

Create a function that will calculate column width (number of characters) for each column in a given table. The function input should be a two dimensional list as the one below:

```
1  [['Name', 'Item', 'Amount', 'Unit_Price', 'Total_Price'],  
2  ['Bettison, Elnora', 'Doxycycline Hyclate', 98, 23.43, 2296.14],  
3  ['McShee, Glenn', 'DROXIA', 27, 33.86, 914.22],  
4  ['Conyard, Phil', 'Nadolol', 44, 12.35, 543.4],
```

90% z Lekce 9

```

7  [ 'WOODISON, ANNIE', 'METOLAZONE', 40, 43.06, 1980.70],
8  ['Woodison, Annie', 'DROXIA', 50, 33.86, 1693.0],
9  ['Skupinski, Wilbert', 'Nadolol', 60, 12.35, 741.0],
10 ['Conyard, Phil', 'WRINKLESS PLUS', 49, 23.55, 1153.95],
11 ['Bettison, Elnora', 'Doxycycline Hyclate', 59, 23.43, 1382.37],
12 ['Skupinski, Wilbert', 'Metolazone', 51, 43.06, 2196.06],
13 ['McShee, Glenn', 'Claravis', 1, 9.85, 9.85]]

```

The output should be a formatted string listing widths for individual columns:

```
~/PythonBeginner/Lesson8 $ python col_widths.py
```

COLUMN	WIDTH
COL 1	18
COL 2	19
COL 3	6
COL 4	10
COL 5	11

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



Simulate path search

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [HOME EXERCISES](#) / [SIMULATE PATH SEARCH](#)

We would like to create a function, that will take two inputs

1. Dictionary called **FILESYSTEM** with further nested dictionaries
2. Path to a value in the dictionary **FILESYSTEM**

90% z Lekce 9

Your program should check, whether the following files represented as dictionary keys can be found under the corresponding paths:

```
1 paths = {'/bin/mkdir' : None,  
2         'lib/init/vars/vars.sh' : None,  
3         '/lib/init/vars.sh' : None,  
4         '/home/documents/reports/report1.xls' : None,  
5         '/home/music/album3/song2.mp3' : None,  
6         '/home/music/album1/song2.mp3' : None,  
7         '/lib/systemd/system/sudo.service' : None  
8         }
```

The program should be built to check **only the absolute paths**. Also it should check whether the path is in an absolute form (starts with '/'). All the paths given above are correct, except the second which is missing '/'.

You should run the program as `main()` function which repeatedly calls the function `file_exists(path)` and record the result returned in the variable `paths`.

Finally the resulting `paths` dictionary should be printed to the terminal.

The program should look up the above files in the dictionary `FILESYSTEM`:

- we should think about a directory as a dictionary linking the directory (key) name to its content (value).
- the value linked to the directory name should be a list of further directories (dictionary) and files (strings)

It would be nicer, if we could save the below dictionary in a text file. Once we have it saved this way, we can read it in our program and pass the content into the function `eval()`. This function will convert the string into a dictionary.

```
#FILESYSTEM  
{ "/" :[{ 'bin': ['echo',  
                'mkdir',  
                'ls',
```

90% z Lekce 9

```

        ]
    },
    {'lib': [ { 'init': ['fstab',
                        'vars.sh',
                        'upstart-job'
                    ]
            },
            {'udev' : ['accelerometer',
                      'ata_id',
                      'cdrom_id'
                    ]
            },
            {'systemd': [ { 'system' : ['sudo.service',
                                        'rsync.service',
                                        'anacron.service'
                                    ]
                        },
                        { 'system-sleep': ['notify-upower.sh'
                                           ]
                        },
                        },
                        'systemd-logind',
                        'systemd-udev',
                        'systemd-localed'
                    ]
            }
        ]
    },
    {'home' : [ { 'documents' : [ {'reports' : []},
                                  'ToDo.txt',
                                  'book.pdf',
                                  'results.pdf'
                                ]
            },
            {'music' : [ {'album1': ['song1.mp3',
                                     'song2.mp3'
                                ]
            }
        ]
    }
}

```

90% z Lekce 9

```

        { 'album2' : [ song1,mp3 ,
                      'song2.mp3'
                    ]
        }
    ]
},
{'pictures' : [ {'holiday' : ['photo1.jpg',
                              'photo2.jpg',
                              'photo3.jpg',
                              'photo4.jpg'
                             ]
                },
                {'trip' : ['photo1.jpg',
                          'photo2.jpg',
                          'photo3.jpg',
                          'photo4.jpg'
                         ]
                }
            ]
}
]
}
]
}

```

Example of running the program:

```

~/PythonBeginner/Lesson6 $ python file_exists.py
{'/bin/mkdir': True,
 '/home/documents/reports/report1.xls': False,
 '/home/music/album1/song2.mp3': True,
 '/home/music/album3/song2.mp3': False,
 '/lib/init/vars.sh': True,
 'lib/init/vars/vars.sh': False,
 '/lib/systemd/system/sudo.service': True}

```

90% z Lekce 9

[Click to see our solution](#)

Fill in contracts

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [HOME EXERCISES](#) / [FILL IN CONTRACTS](#)

Create a program that will generate employee contracts based being given the following inputs:

1. Employee data
2. Contract type

There are three different contract types and therefore we need to give our program the name of the template it should populate. The program should be intelligent enough to know, what data should be populated into which contract type.

Employee Data

Paste the following dictionary into a text file created using text editor and save it under **employees.txt**. You should remember the place where you store it, because you will need to open it in your program.

```
1 {'X12345' : {'ID': 'X12345', 'full_name': 'Jack Frank',
2             'birthdate': '1.1.1970' , 'job_title' : 'welder',
3             'position_from': '1.5.2015', 'contract_start': '1.2.2013',
4             'contract_end': '31.12.2020', 'salary': 123456},
5 'X54321' : {'ID': 'X54321', 'full_name': 'Bob Doe',
6             'birthdate': '8.8.1971' , 'job_title' : 'machinist',
7             'position_from': '1.8.2016', 'contract_start': '1.8.2014',
8             'contract_end': '31.12.2021', 'salary': 23451}}
```

90% z Lekce 9

stored in a separate directory Note that the templates need to be adjusted in order Python fill the data into correct places in the document:

salary_change.txt

```
1 Company & CO
2 Main Street, 5
3 Newyorkshire
4 Somewhere
5
6             SALARY CHANGE
7
8
9 Employee {full_name}, (ID: {ID}) born on {birthdate},
10 has agreed to accept change of the salary amounting to
11 {salary} dollars.
12
13 The new salary shall be paid with the next payroll.
14
15 Signature
16
17 _____
18 {full_name}
```

job_change.txt

```
1 Company & CO
2 Main Street, 5
3 Newyorkshire
4 Somewhere
5
6             JOB CHANGE
7
8
9 Employee {full_name}, (ID: {ID}) born on {birthdate},
```


90% z Lekce 9

```
12 This change is valid as of {position_from}
13
14 Signature
15
16 _____
17 {full_name}
```

contract_prolongation.txt

```
1 Company & CO
2 Main Street, 5
3 Newyorkshire
4 Somewhere
5
6             CONTRACT PROLONGATION
7
8
9 Employee {full_name}, (ID: {ID}) born on {birthdate},
10 hereby agrees to change the employment contract
11 termination to {contract_end}.
12
13 This change is performed with immediate validity.
14
15 Signature
16
17 _____
18 {full_name}
```

Example of running the program:

```
~/PythonBeginner/Lesson6 $ python fill_contracts.py
Please select the option number of action you want to perform:
0. salary change
1. job change
2. contract prolongation
0
```

90% z Lekce 9

`Contracts have been generated.`

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



Mortgage [H]

[PYTHON ACADEMY](#) / [8. STRING FORMATTING & TEXT FILES](#) / [HOME EXERCISES](#) / [MORTGAGE \[H\]](#)

Everybody, who is going to take a mortgage would like to know, whether the payments and interests are calculated well.

There is some obscure magic behind the calculation and it is hard to reverse engineer the way, how our bank really calculated the payments.

Fortunately, we can make our own calculator, that will print nicely our monthly payments.

Or maybe we would like to know, what is the total amount of money, we will have to pay to our bank. Our calculator should do that as well.

Your task will be to create a program that will print out the following table decomposing the mortgage into individual monthly payments:

Loan: 2000000 Years: 30 Interest: 693880 Monthly Payment: 7483			
=====			
	Payment		Interest
			Principal
			Left to Pay
=====			
	1		4,692
			2,791
	2		4,679
			2,804
	3		4,666
			2,817
			2,693,880
			2,686,397
			2,678,914

90% z Lekce 9

	6		4,627		2,856		2,656,465	
.....								
.....								

The inputs into the program are:

1. The amount of money lent = Loan
2. Number of years in which the mortgage will be paid = years
3. Monthly interest rate offered by the bank (e.g. 2.09%)

In order we can present the above output, the program will then have to calculate/generate:

1. Monthly payment we will have to send to our bank:

$$\text{monthly_payment} = L * (r * (1 + r)^n) / ((1 + r)^n - 1)$$

where:

L - is the amount of money we are borrowing

r - is the interest rate that bank offers us (e.g. 2.09%) - should be converted into decimal number
=> 0.0209 and divided by 12 as this is yearly interest rate: $r = \text{yearly_rate} / 100 / 12$

n - is the number of months in which we will pay our mortgage (when talking to the bank, we say
- I want it for 20 years for example - they convert it into number of months)

2. Total amount of money that we will have to pay to the bank

$$\text{total_amount} = \text{monthly_payment} * \text{number_of_months}$$

3. Total interest, that we will pay to the bank

$$\text{total_interest} = \text{total_amount} - \text{loan}$$

4. The listing of individual monthly payments decomposed into

- **payment number** (from 1 until the last month's number)

90% z Lekce 9

```
monthly_interest = L * r
```

where:

L - is the amount of money that is left to be paid

r - is the monthly interest rate converted from the yearly interest rate that bank offers us (e.g. 2.09% -> 2.09 / 100 / 12)

- **principal part** = the amount of money we pay to actually lower our debt

```
principal = monthly_payment - monthly_interest
```

- **the total amount of money to be still paid**

At the beginning interest part is high, and principal quite low. However, with each payment, the debt is lowered every month and monthly interest rate is lower and lower, the monthly payment is constant and therefore principal is higher and higher.

Example of running the program:

```
~/PythonBeginner/Lesson6 $ python mortgage.py
How much do you want to borrow?: 2000000
At what rate?: 2.09
How many years?: 30
| Loan: 2000000 | Years: 30 | Interest: 693880 | Monthly Payment:
7483 |
=====
|      Payment      | Interest | Principal | Left to Pay |
|=====|=====|=====|=====|
|          1         |   4,692  |   2,791   | 2,693,880   |
|=====|=====|=====|=====|
|          2         |   4,679  |   2,804   | 2,686,397   |
|=====|=====|=====|=====|
|          3         |   4,666  |   2,817   | 2,678,914   |
```

90% z Lekce 9

5	4,640	2,843	2,663,948
6	4,627	2,856	2,656,465
7	4,614	2,869	2,648,982

We will divide the following solution section into two subsections:

- The first one is dedicated to the mortgage payments calculation logic,
- the other is related to table formatting.

Calculation Logic Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



Formatting Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution



DALŠÍ LEKCE