Osnova

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.

100% z Lekce 9

Osnova

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,

For more information on the subject, please see the Wikipedia page

Osnova

Examples of years:

- years 1600, 2000 are leap, because they are divisible by 4 and even through 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
```

100% z Lekce 9

Osnova

spustit kód

Code Solution

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

Click to see our solution

- This function decides, whether the input integer represents a leap year or not.
- Instead of writing the entire if-else condition statement, we can just write the test, that would be otherwise evaluated inside the if clause
- The condition can be expressed in words as follows:

  "The value of year is divisible by 400 or it is divisible by 4 and not by 100"

The **longer version** with if statement:

```
1 def is_leap(y):
2     if ((y % 400 == 0) or (y % 4 == 0 and y % 100 != 0)):
3         return True
4     else:
5         return False
```

The **shorter version** without the conditional statement:

```
1 def is_leap(y):
2     return ((y % 400 == 0) or (y % 4 == 0 and y % 100 != 0))
```

100% z Lekce 9

Osnova

ays

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:

```
>>> num_days((2017,1,1),(2017,3,4))
62
```

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

We have defined 3 functions:

Osnova

Function num_days()

This function is the one that we run as the main function. It should take two arguments - dates represented as tuples and return an integer representing the number of days between the two dates.

The main purpose of this function is to increase the counter with each increase of date represented by `date1`. That way we gradually approach date `date2`. Once `date1 == date2`, the counter is stopped and the result is returned.

The date increase is performed by a helper function `increase()` - see next.

```
def num_days(date1, date2):
    if date1 > date2:
        date1, date2 = date2, date1

    num_days = 0
    while date1 < date2:
        date1 = increase(date1)
        num_days += 1
```

Function increase()

Task of this function is to take in an input in form of tuple or a list and decide, which item of the input object should be modified. To code this function, it is better to think first, what manipulations will we want to perform to items in the input tuple/list:

1. **increase only the day item** - the operation is `(y,m,d+1)`, e.g. input is `(2017,3,4)`, output should be `(2017,3,5)`

2. **change the month** - the operation is `(y,m+1,1)`, e.g. input is `(2017,3,31)`, output should be `(2017,4,1)`. The month item has to be increased by one and day has to be reset to value 1

100% z Lekce 9

We decided to handle February in a separate condition

Osnova

- In general, we want to check, whether the value of month item already reached 30 for months `[4,6,9,11]` or 31 for months `[1,3,5,7,8,10]`
- if month 2 reached day 28 or 29 we need to check, whether the year is leap or not

3. **change the year** - the operation is `(y+1,1,1)`, e.g. input is `(2017,12,31)`, output should be `(2018,1,1)`. The year item has to be increased by one and both month and day have to be reset to value 1.

```
def increase(date):

    y,m,d = date

    # February
    if m == 2 and d in [28,29]:
        if is_leap(y) and d == 28:
            date =(y,m,d+1)
        else:
            date =(y,m+1,1)

    # month end
    elif (m in [1,3,5,7,8,10] and d == 31) \
        or (m in [4,6,9,11] and d == 30):
        date =(y,m+1,1)

    # year end
    elif m == 12 and d == 31:
        date = (y+1,1,1)

    #in month
    else:
        date = (y,m,d+1)

    return date
```

Function is_leap()

100% z Lekce 9

Osnova

...of writing `if-else` condition statement, we can just write the test, that would ...side the `if` clause

- The result of this condition is `True` or `False` and that is exactly what we want to know
- The condition can be expressed in words as follows: "The value of year is divisible by 400 or it is divisible by 4 and not by 100"

```
def is_leap(y):
    return ((y % 400 == 0) or (y % 4 == 0 and y % 100 != 0))
```

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

Click to see our solution

Again, we have therefore 3 functions:

- `num_days()`
- `days_in_year()`
- `is_leap()`

This solution uses again function `is_leap()`. It also relies on the principle on increasing a value, in this case the value of the year of the first date, thus approaching the year of the second date.

Function num_days()

Function `num_days()` in this case keeps adding number of days in each year to the `days` counter variable. This function heavily relies on the logic implemented in `days_in_year()` function.

The main task of `num_days()` here is to assure that `days_in_year()` receive the correct inputs:

1. while the year value of the first date is smaller than the end date's year, function `_____()` receives the second argument as the last day of that year. Subsequently we move the start date's values to the 1st of January of the following year. Note that we had to add `+1` days in this case, because we have artificially changed the first date to the first of January instead of the 31st of December.
2. once the year1 reaches the value of year2, we just calculate number of days left in that last year, again using `days_in_year()`. Then we break out of the loop

```
def num_days(date1,date2):
    if date1 > date2:

        date1, date2 = date2, date1

    y1,m1,d1 = date1
    y2,m2,d2 = date2

    days = 0

    while y1 <= y2:
        if y1 < y2:
            days += days_in_year((y1,m1,d1), (y1,12,31)) + 1
            y1,m1,d1 = y1+1,1,1
        else:
            days += days_in_year((y1,m1,d1), (y2,m2,d2))
            break

    return days
```

Function days_in_year()

This function always calculates the **number of days** between two dates **in the same year**. We have to distinguish two cases:

1. `date1` and `date2` belong to the same month, that means `m1 == m2` - in that case is just enough to make the difference `d2 - d1`.
2. Otherwise we need to sum the number of days in months between `m1` and `m2` (`sum(months[m1:m2-1])`) and add these to the number of the days of from the first

```
[m1-1] - d1) + sum(months[m1:m2-1]) + d2
```

Osnova

Here also, we need to check, whether the first date falls before the February 28th. In that case we need to add 1 day to the result if the year is leap.

```
if (m1,d1)<=(2,28): days += is_leap(y1)
```

Instead of a loop, we just used the built-in `sum` function.

```
def days_in_year(date1, date2):
    y1, m1, d1 = date1
    m2, d2 = date2[1:]
    months = (31,28,31,30,31,30,31,31,30,31,30,31)

    if m1 == m2:
        days = d2 - d1
    else:
        days = (months[m1-1] - d1) + sum(months[m1:m2-1]) + d2

        if (m1,d1)<=(2,28): days += is_leap(y1)

    return days
```

Function is_leap()

```
def is_leap(y):
    return ((y % 400 == 0) or (y % 4 == 0 and y % 100 != 0))
```

100% z Lekce 9

Osnova

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:

100% z Lekce 9

```
2  ...int(...put('How many people live in your city? '))
3  ...'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)
```

Try it:

```
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

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,
2. sets or
3. dictionary views

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

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

```
employee_data = {'name': "John", 'surname': 'Smith', 'salary':1200,
'job':'Manager'}
template = '| {surname} | {name} | {salary} | {job} |'
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:

```
data = [['Company',     'Date',       'Amount'],
        ['Food Store', '2017-08-01',  -112.56],
```

```
5       ['C   es&CO',    '2017-08-09',  -2130.321],
6       [     ctro',     '2017-08-15',  -1566.932],
7       ['Mother',       '2017-08-22',   5000.00]]
```

Osnova

Field Width

Let's say, we want all our table columns to be of the same width. The width provided for each field is 8 characters. To tell this to Python, we can create a string to be formatted first:

```
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

100% z Lekce 9

me ... precision rules:

Osnova

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

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

Code Task

Print each row of the table providing each column with the width of 8 characters, aligning the first 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

Osnova

`+` symbol there. We will force the values in the last column to show both

```
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:

```
row_template = '| {:<8.8} | {:_^12} | {:>+8,.6} |'
```

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

100% z Lekce 9

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

Osnova

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

Opening the File with Python

Before performing any operations with files in our programs, we need to first create their representation in form of **file object**. File objects is created using the built-in function `open()` .

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:

```
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:

```
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:

```
file.close()
```

100% z Lekce 9

Osnova

Close the file.

Writing into the File

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

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

We use flag '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

100% z Lekce 9

```
Osnova  ine number: 5
```

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

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

In order to write and read what has been written, we need to be able to move the our position in the file - we need to navigate it.

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.

Osnova

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

```
This is the line number: 9
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

Osnova

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       |
```

Osnova

We ... to add using `+` operator to concatenate strings:

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

100% z Lekce 9

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

Osnova

Format method

Methods are called on objects, in this case a string object. Methods are associated with the object via the `.` sign.

```
>>> 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.

100% z Lekce 9

There are two conversion specifiers, but at the moment we can get by with two conversion ones.

Osnova

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

Python uses these as placeholders to which the values from the tuple will be placed. 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)
```

Osnova

ertion order

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:

```
```

```
  F...     ...    line 1, in <module>
        Osnova
...        ...   index out of range
```

Insertion by key

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
```

100% z Lekce 9

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

Osnova

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

```
>>> name = 'Bob'
>>> age = 37
>>> 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:

100% z Lekce 9

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

Value Formatting

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 formated text: {[keyname_or_index]:[[fill] align] [sign] [#] [0] [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**.
- **align** - Possible values - `<`, `>`, `^` - left alignment / right alignment / centered alignment
- **sign** -Possible value `+`, `-` or `" "` (space) - force sign (+ or minus) and if needed insert 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** - referes to flags we already know such as `f` (float) or `d` (decimal)

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 "|"

We can set that field to 2 characters long fills the empty space by space character `" "`.

Osnova

Alignment and Fill

PYTHON ACADEMY / 8. STRING FORMATTING & TEXT FILES / STRING FORMATTING / ALIGNMENT AND FILL

Once we know how to set field width, we can specify, what character will be used to fill blank spaces. The default fill is the white space `" "` character, but we can be more creative here. We 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)
```

100% z Lekce 9

Osnova

Precision tells python, that 5 digits are allowed. In this case the number had to be rounded to 2 decimals because the total number of digits is be 5.

We can play with the precision value to see, how the number inside the string is depcited. 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)
'|1.23e+02|123.5|123.46|'
```

We can now combine other specifiers with precision:

```
>>> '|{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

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)
```

Osnova

```
>>> '|{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.

Forcing Signs

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

100% z Lekce 9

Osnova	Description	Example	Output
`.ljust(num,'char')`	Lets us to specify the 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.	`"5".ljust(5,"x")`	`'5xxxx'`
`.rjust(num,'char')`	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.	`"5".rjust(5,"x")`	`'xxxx5'`
`.center(num,'char')`	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.	`"5".center(5,"x")`	`'xx5xx'`

Osnova	Description	Example	Output
`.zfill(num,'char')`	Lets us to specify 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.	`"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 +

100% z Lekce 9

Osnova

xpression Specifiers

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

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

- **keyname** - we have alread 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

```
>> '|%-4,)
    Osnova
```

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

Field width specified by an integer:

Osnova

```
>>> '|%#6d|%06d|%-6d|%+6d|% 6d|' % (45, 45, 45, 45, 45)
'|    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 inptu 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

100% z Lekce 9

```
>>> num = 1.123457
>>> '|%#6.*d|%06.*d|%-.*d|%+6.*d|% 6.*d|' % (num, num, num, num, num)
'|1.12346|1.123457e+00|1.12346|'
```

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

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

string formatting is to implement it even on actual formatting parameters

Osnova

```
>>> '|{:{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**

100% z Lekce 9

The following examples demonstrate conversion from decimal (method input) into binary (flag `b`), octal (flag `o`) and hexadecimal (flag `x`) representation.

```
>>> '|{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)
'|Ę*****|'
```

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?

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.

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`

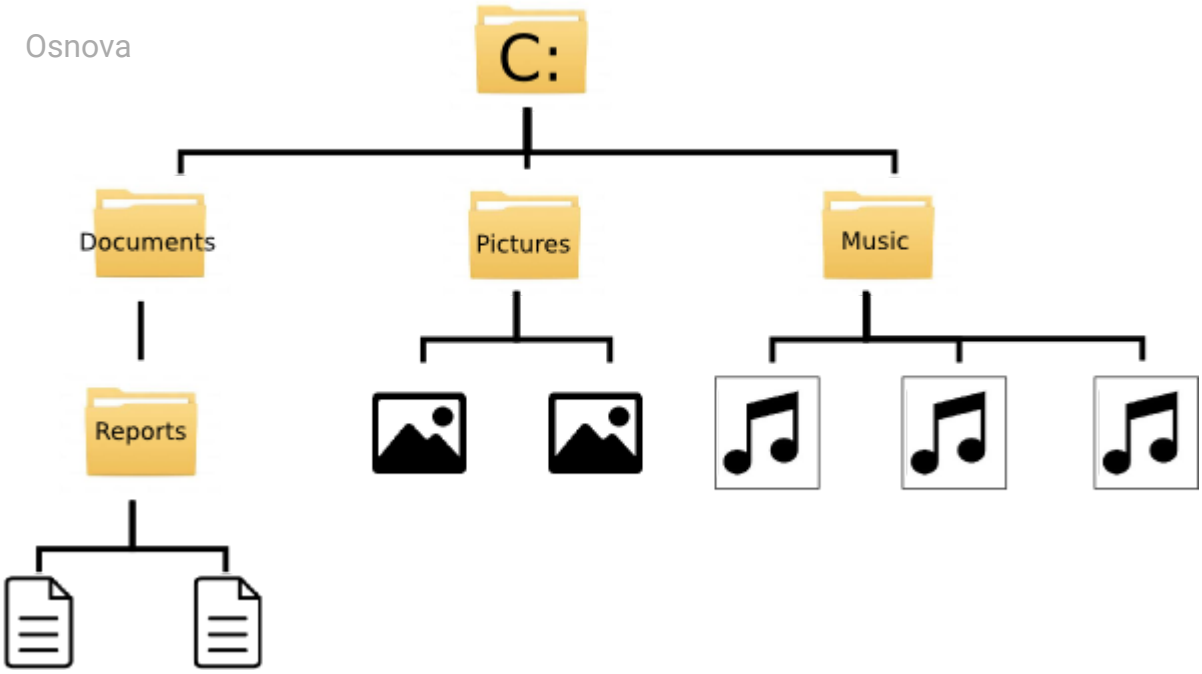
We can see, that file paths are written with backslash `'\'` in Windows and forward slash `'/'` in Linux or Mac OS (let's call them both Unix) separating folders. They specify the starting point (the leftmost name) and the end point (the rightmost name). Starting point usually refers to a folder name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



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.

If we [illegible] Python script `invert_image.py` located in `/home/Bob/Scripts/`, the current working directory is the directory `Scripts`. Now let's say, that the folder `Scripts` contains 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

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.

1. Osnova file called `test.txt` in your `PythonBeginner/Lesson6` directory and lowing text:

```
This is the first line
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'
```

100% z Lekce 9

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

Osnova

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.
```

W

100% z Lekce 9

Osnova

to a file, we need to **open it in writing mode**. We tell Python to open a file 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

100% z Lekce 9

Osnova

we read the file content, we cannot read it once again unless we close it and open it again. To avoid reopening the file object just in order to re-read its content, file object 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'
```

100% z Lekce 9

...ca... ...file ...ain.

Osnova

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.

Osnova

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

100% z Lekce 9

Files opened with the `'r+'` allow for writing and reading. At the same time, the file content is not removed when the file is opened. The data begins to be written from the first position in the 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

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')
```

Mode [illegible] not support reading operation:

Osnova

```
>>> file.read()
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

Osnova

within the `open()` function as the second argument. We can distinguish ly vs. read and write modes.

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

Osnova

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 ;)

Osnova **f reading files**

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']
```

100% z Lekce 9

We can specify how many lines we want to read, by supplying the method with an integer. Osnova
Jv ily e line:

```
>>> 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 it usage with only one input, which specified the index to which we want to move the file cursor.

```
relative_to)
```

Osnova

Offset relative to the file beginning

Offset relative to the beginnning 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

100% z Lekce 9

Osnova

```
>>> 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

Osnova

...the size of a file, we need to import the module called `sys` and use its ...:

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

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

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

Osnova

QUIZ

String Formatting

1/15

Which formatting expression is correct?

100% z Lekce 9

Osnova

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

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

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

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

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

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:

```
[['Name', 'Item', 'Amount', 'Unit_Price', 'Total_Price'],
['Bettison, Elnora', 'Doxycycline Hyclate', 98, 23.43, 2296.14],
['McShee, Glenn', 'DROXIA', 27, 33.86, 914.22],
['Conyard, Phil', 'Nadolol', 44, 12.35, 543.4],
['Bettison, Elnora', 'Claravis', 91, 9.85, 896.35],
['Idalia, Craig', 'Nadolol', 83, 12.35, 1025.05],
['Woodison, Annie', 'Metolazone', 46, 43.06, 1980.76],
```

```
[illegible]d, [illegible]il', 'WRINKLESS PLUS', 49, 23.55, 1153.95],
[illegible]on, [illegible]lnora', 'Doxycycline Hyclate', 59, 23.43, 1382.37],
['Skupinski, Wilbert', 'Metolazone', 51, 43.06, 2196.06],
['McShee, Glenn', 'Claravis', 1, 9.85, 9.85]]
```

Osnova

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

If you are confident enough, you can skip all the way down to see the whole solution. If you are not, or if you had some trouble with this task, you can read through step-by-step solution.

Before writing any code, lets find out what we need to complete our task. We need:

1. Count number of all characters in each row and store that information
2. Print table with results
    - Optional: Print nice table

First, we need to have a **list of widths** which we're going to continually update. The list will contain the lenghts of each column. Then we use the nested for loop from above to get to the lenght of each word and compare it with the lenght we already have saved in our list.

```
def GetWidths(big_list):
    widths = [0]*len(big_list[0])
```

Osnova

Next, our dataset is a **list of lists.** Inside the nested list are characters we want to compare. So we need to step through both of the lists - we will need to use two **for** loops (nested for loop).

```
for row in big_list:
    for word in row:
        #do something
```

Then, we know that number of characters in column is given by lenght of the longest word in that column. We, therefore, **compare** the length of the item with the width value we already have in our width list.

If the item lenght is bigger we **replace** the width with the item lenght. Note that we use the variable `i` which helps us to match the position of the width in its list and the item we're comparing. `i` is always increased by 1 in each **inner** loop. When we get to the next row, `i` is set again to 0.

When determining the lenght of a word, we need to ensure, that the word is really a **string**, so we don't get errors.

```
for row in big_list:
        i = 0
        for word in row:
            if len(str(word)) > widths[i]:
                widths[i] = len(str(word))
            i += 1
```

And that's it! Now we need to create a function from that. This is pretty easy. We have one and only input - the big list and we need to return the list with stored column lenghts.

```
def GetWidths(big_list):
    widths = [0]*len(big_list[0])

    for row in big_list:
        i = 0
        for word in row:
            if len(str(word)) > widths[i]:
```

100% z Lekce 9

```
    tur  widths
```

Osnova

Now we need to create the printing function. This function will have two parameters:

1. List with column lenghts
2. Some header which will be printed first - we can set a default one

```
def PrintWidthsTable(widths, header=['COLUMN','WIDTH']):
```

First we need to print the header, but we also need some lines - this can easily by done with *pipes* `|` . First we need to create the table layout and then fill in the content - here comes handy the `.format` method.

```
def PrintWidthsTable(widths, header=['COLUMN','WIDTH']):
    firstrow = '| {} | {} |'.format(*header)
    print(firstrow)
```

Now the harder part. We need to print the rest of table which contains the information we want. We need to:

1. Print string `COL` with number of the column.
2. Print actual column lenght.

We will print each column separately and again we will need to step through the list which stores the column lenghts, so again we will use a **for** loop. We don't have stored information about column number anywhere, but it does not matter. We will use function `enumerate` to get that information. This will be done in the **for** statment so beware, we need to unpack two walues - column number and column lenght.

```
for num, width in enumerate(widths):
```

Printing a row with information is similar to printing the first row. First we will create the appearence and than fill that with information through the `.format` method. And finally print the row.

```
for num, width in enumerate(widths):
    row = '| COL {} | {} |'.format(num, width)
    print(row)
```

100% z Lekce 9

Osnova

```
        print('this', widths)
        firstrow = '| {} | {} |'.format(*header)
    print(firstrow)

    for num, width in enumerate(widths):
        row = '| COL {} | {} |'.format(num, width)
        print(row)
```

And that's it! The output will look like this:

```
| COLUMN | WIDTH |
| COL 0  |  18   |
| COL 1  |  19   |
| COL 2  |   6   |
| COL 3  |  10   |
| COL 4  |  11   |
```

Which is good, it presents information we wanted, but it's not a nice thing to look at. Let's change that! The problem is that the widths of columns of our printed table are variable - we need to fix that. We need to set width of column to a fix number. This can be done by improving our formating. To the curly brackets we will add `:^` and the number. This will center the text in column. We also need to pick the number. This is quiet easy too - because the width of the column must respond to lenght of longest word that will be writen into the column. In our example the longest words for both columns are the header words. SO now we need to upgrade our row template to adopt this.

```
for num, width in enumerate(widths):
        #COLUMN = 6 letters, WIDTH = 5 letters
        row = '| COL {:^6} | {:^5} |'.format(num, width)
        print(row)
```

This will create nice output, where lines are connected. But if you change the header, or if you input list with lot of columns, or list with columns that contain lot of stuff, this solution will not work. We need to set the column width dynamicaly. This will requie some additional code. We need to solve these task to get to the solution:

1. How to get the column width
2. How to dynamicaly put that number into our row template.

ty[illegible] ur column. The header word (which can be cahnged) and `COL +` Osnova enght of the header word is clear, but how to get lenght of `COL +` **number**, when we don't know how many colums will be in our analyzed list? This is simple too after all. Because number of colums responds to number of column lenght values. And those are stored in a list. So lenght of this list responds to number of colums. But beware, we don't need the lenght of that list. We need to know how many ciphers has the lenght of that list - how many characters will the lenght (the number) occupy. This can be easily computed by converting the lenght number to **string**, and then determining lenght of that string. To get the width of `COL` + **number**, you just add **4** (COL + space = 4 characters) to lenght of this string. Now we need to decide what is longer. Header word or `COL` + **number** stuff? This decision can be made throught **if** statement, but it's better to use function `max`. This function returns the greatest of given arguments.

```
def PrintWidthsTable(widths, header=['COLUMN','WIDTH']):
    c1 = max(len(header[0]), len(str(len(widths))) + 4)
```

The second column is easier. We need to decide beteween lenght of header and number of ciphers of bigest column lenght. To get the second number we just need to find the biggest number in list with column lengts. It may look like a difficult task, but we just simply need to sort that list and then grab the last value. Sorting list is done via function `sorted` and geting cipher count is done again by converting the number to string and then determining lenght of that string. And again the whole decision process will use the `max` function.

```
def PrintWidthsTable(widths, header=['COLUMN','WIDTH']):
    c1 = max(len(header[0]), len(str(len(widths))) + 4)
    c2 = max(len(header[1]), len(str(sorted(widths)[-1])))
```

Now how to get those numbers into our row template? Because we cannot simply smash those variables into the template string. But we can cut the template to pieces, insert the numbers and than again glue it together. Beware now we also need to change the first row template. The template string will look like this:

```
'| {0:^'+ str(c1) +'} | {1:^' + str(c2) + '} |'
```

And whole function together:

```
def PrintWidthsTable(widths, header=['COLUMN','WIDTH']):
    c1 = max(len(header[0]), len(str(len(widths))) + 4)
    c2 = max(len(header[1]), len(str(sorted(widths)[-1])))
```

Osnova

```
    for num, width in enumerate(widths):
        row = '| {0:^'+ str(c1) +'} | {1:^' + str(c2) + '} |'
        row = row.format('COL ' + str(num), width)
        print(row)
```

And the output will lool like this:

```
| COLUMN | WIDTH |
| COL 0  |  18   |
| COL 1  |  19   |
| COL 2  |   6   |
| COL 3  |  10   |
| COL 4  |  11   |
```

Whole solution

```
LISTS = [['Name', 'Item', 'Amount', 'Unit_Price', 'Total_Price'],
['Bettison, Elnora', 'Doxycycline Hyclate', 98, 23.43, 2296.14],
['McShee, Glenn', 'DROXIA', 27, 33.86, 914.22],
['Conyard, Phil', 'Nadolol', 44, 12.35, 543.4],
['Bettison, Elnora', 'Claravis', 91, 9.85, 896.35],
['Idalia, Craig', 'Nadolol', 83, 12.35, 1025.05],
['Woodison, Annie', 'Metolazone', 46, 43.06, 1980.76],
['Woodison, Annie', 'DROXIA', 50, 33.86, 1693.0],
['Skupinski, Wilbert', 'Nadolol', 60, 12.35, 741.0],
['Conyard, Phil', 'WRINKLESS PLUS', 49, 23.55, 1153.95],
['Bettison, Elnora', 'Doxycycline Hyclate', 59, 23.43, 1382.37],
['Skupinski, Wilbert', 'Metolazone', 51, 43.06, 2196.06],
['McShee, Glenn', 'Claravis', 1, 9.85, 9.85]]


def GetWidths(big_list):
    widths = [0]*len(big_list[0])

    for row in big_list:
```

100% z Lekce 9

```
                widths[i] = len(str(word))
            i += 1
    return widths


def PrintWidthsTable(widths, header=['COLUMN','WIDTH']):
    c1 = max(len(header[0]), len(str(len(widths))) + 4)
    c2 = max(len(header[1]), len(str(sorted(widths)[-1])))

    firstrow = '| {0:^'+ str(c1) +'} | {1:^' + str(c2) + '} |'
    firstrow = firstrow.format(*header)
    print(firstrow)

    for num, width in enumerate(widths):
        row = '| {0:^'+ str(c1) +'} | {1:^' + str(c2) + '} |'
        row = row.format('COL ' + str(num), width)
        print(row)

widths = GetWidths(LISTS)
PrintWidthsTable(widths)
```

Osnova

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`

The task of the function is to tell, whether such a value encounters itself in the dictionary `FILESYSTEM`

Your program should check whether the following files represented as dictionary keys can be

100% z Lekce 9

```
Osnova
    '/bin/mkdir' : None,
    '/init/vars/vars.sh' : None,
    '/lib/init/vars.sh' : None,
    '/home/documents/reports/report1.xls' : None,
    '/home/music/album3/song2.mp3' : None,
    '/home/music/album1/song2.mp3' : None,
    '/lib/systemd/system/sudo.service' : None
    }
```

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 resuling `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',
                  'ip',
                  'kill'
                 ]
        },
```

```
                        'upstart-job'
                       ]
                },
                {'udev' : ['accelerometer',
                           'ata_id',
                           'cdrom_id'
                           ]
                },
                {'systemd': [  { 'system' : ['sudo.service',
                                             'rsync.service',
                                             'anacron.service'
                                            ]
                                  },
                                  { 'system-sleep':['notify-upower.sh'
                                                   ]
                                  },
                                  'systemd-logind',
                                  'systemd-udevd',
                                  'systemd-localed'
                                ]
                }
              ]
        },
       {'home' : [ { 'documents' :[ {'reports' : []},
                                    'ToDo.txt',
                                    'book.pdf',
                                    'results.pdf'
                                  ]
                   },
                   {'music' : [ {'album1': ['song1.mp3',
                                            'song2.mp3'
                                            ]
                                  },
                                 {'album2' : ['song1,mp3',
                                              'song2.mp3'
```

Osnova

100% z Lekce 9

```
                                ]
        Osnova
                        },
                        {'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}
```

Code Solution

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

100% z Lekce 9

Click to see our solution

Osnova

In this program, we have decided to declare the variable paths as a global variable.

```
paths = {'/bin/mkdir' : None,
         'lib/init/vars/vars.sh' : None,
         '/lib/init/vars.sh' : None,
         '/home/documents/reports/report1.xls' : None,
         '/home/music/album3/song2.mp3' : None,
         '/home/music/album1/song2.mp3' : None,
         '/lib/systemd/system/sudo.service' : None
         }
```

Function main()

The main function checks for every entry in the `paths` variable, whether the given path can be found in the file system. It then record the result as the value of the corresponding key in this dictionary.

At the end we use this special import statement that makes use of pretty print library to print our dictionary in a nice way. We could, use a for loop as well.

```
def main()

    with open('FILESYSTEM.txt') as file:
        filesystem = eval(file.read())

    for path in paths:
        paths[path] = file_exists(path,filesystem)

    from pprint import pprint as pp
    pp(paths)
```

Function file_exists()

This is the main step of this program. Here we split the path string into separate nodes

100% z Lekce 9

We [illegible] to register so called **current working directory** - cwd. For each valid node in the [illegible] directory is changed to the content of that node - dictionary. Thus we gradually reduce the size of the structure in cwd and we approach the end of the tree.

We check, whether a given folder or file encounters itself in cwd, using `search_folder(cwd,node)` function. This function returns None in case that the searched node has not been found in cwd. That causes the code execution to break out of the loop and return False from the `file_exists()` function.

The `target`, we are searching for is the last node in the path.

```
def file_exists(path,filesystem):
    nodes = split_path(path)
    target = nodes[-1]

    cwd = filesystem['/']

    while nodes:
        node = nodes.pop(0)
        cwd = search_folder(cwd,node)

        if cwd == target and not nodes:
            return True
        elif not cwd:
            break

    return False
```

Function search_folder()

This is a helper function to `file_exists()`. It just iterates over the dictionary `folder` and checks, whether the next item in the folder is a folder and its name corresponds to the searched name, or it is a file (string) and in that case the file is returned.

If the for loop terminates without meeting any of the above conditions, the function returns value `None` by default.

```
def search_folder(folder, target):
```

```
            return item[target]
        if item==target:
            return item
```

Osnova

Function split_path()

This function returns the nodes listed on the path. It checks whether the given absolute path is correctly written (starts with '/').

```
def split_path(path):
    if not is_absolute(path):
        return path.split('/')
    else:
        return path[1:].split('/')
```

Function is_correct_absolute()

This function tells us, whether a given string contains a correct absolute path (begins with root `'/'`) or not.

```
def is_absolute(path):
    return path.startswith('/')
```

Whole Solution

```
paths = {'/bin/mkdir' : None,
         'lib/init/vars/vars.sh' : None,
         '/lib/init/vars.sh' : None,
         '/home/documents/reports/report1.xls' : None,
         '/home/music/album3/song2.mp3' : None,
         '/home/music/album1/song2.mp3' : None,
         '/lib/systemd/system/sudo.service' : None
         }

def main()

```

100% z Lekce 9

```
14
    Osnova      in paths:
16          paths[path] = file_exists(path,filesystem)
17
18      from pprint import pprint as pp
19      pp(paths)
20
21
22  def file_exists(path,filesystem):
23      nodes = split_path(path)
24      target = nodes[-1]
25
26      cwd = filesystem['/']
27
28      while nodes:
29          node = nodes.pop(0)

30          cwd = search_folder(cwd,node)
31
32          if cwd == target and not nodes:
33              return True
34          elif not cwd:
35              break
36
37      return False
38
39
40  def search_folder(folder, target):
41      for item in folder:
42          if isinstance(item,dict) and target in item:
43                  return item[target]
44          elif item==target:
45              return item
46
47
48  def split_path(path):
49      if not is_absolute(path):
50          return path.split('/')
51      else:
```

100% z Lekce 9

```
54
    Osnova ...ute(path):
56      return path.startswith('/')
57
58
59  main()
```

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'
```

Osnova

Please store the following text templates under the corresponding names. Templates can be 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

```
Company & CO
Main Street, 5
Newyorkshire
Somewhere

            SALARY CHANGE


Employee {full_name}, (ID: {ID}) born on {birthdate},
has agreed to accept change of the salary amounting to
{salary} dollars.

The new salary shall be paid with the next payroll.

Signature

_____________
{full_name}
```

job_change.txt

```
Company & CO
Main Street, 5
Newyorkshire
Somewhere

            JOB CHANGE

```

```
to change the job title to {job_title}.

This change is valid as of {position_from}

Signature

_____
{full_name}
```

contract_prolongation.txt

```
Company & CO
Main Street, 5
Newyorkshire
Somewhere

            CONTRACT PROLONGATION


Employee {full_name}, (ID: {ID}) born on {birthdate},
hereby agrees to change the employment contract
termination to {contract_end}.

This change is performed with immediate validity.

Signature

_____
{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
```

```
Osnova
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

Our program will have to solve the following questions:

- Where to find all the templates and where to store the generated contracts?
- Where to get the employee data from?
- How should the program interact with the user?
- How to open the template and how to create each new contract file?

The first few lines of code contain variable names all in uppercase. This in Python normally means, that we want to treat the data refered by such variables as constants - we will not change them in the future.

These constants serve as a kind of a configuration of our program. Therefore, it is better to have them at the top.

These lines solve the problem:

Where to find all the templates and where to store the generated contracts?

It also tells Python, where the file will all the employee data is stored (`EMPLOYEE_DB_PATH`).

```
1 TEMPLATE_PATH = '/home/martin/PythonBeginner/Lesson6/Templates'
2 CONTRACTS_PATH = '/home/martin/PythonBeginner/Lesson6/Contracts'
3 EMPLOYEE_DB_PATH =
  '/home/martin/PythonBeginner/Lesson6/employees.txt'
4
```

Th[illegible]ve only as an example. You will probably have different folders on your [illegible] and your program could have in common is the folder name for template and contracts.

Osnova

Function main()

The `main()` function serves to coordinate the program. It performs the following actions:

- prompt user to choose, what template will be used (`input(print_prompt())`)
- load the employee data (`load_db(EMPLOYEE_DB_PATH)`)
- load the template (`load_template(template_name)`)
- generate individual contract files (`write_file(employee, template, template_name)`)

Each of these actions is represented by a function call. These functions we have implemented below.

It is also important to note the heavy use of `format()` method to dynamically create strings in the program.

```
def main():
    num = int(input(print_prompt()))

    employees = load_db(EMPLOYEE_DB_PATH)

    template_name = TEMPLATES[num].replace(' ', '_')
    template = load_template(template_name)

    for employee_id, employee in employees.items():
        print('Creating contract for %s ....'%(employee_id,))
        write_file(employee, template, template_name)


    print('\nContracts have been generated.')
```

Function print_prompt()

Wh[illegible]ctir[illegible]t returns a newline character, which is then printed by the `input()` [illegible] we have called `print_prompt()`


```
def print_prompt():
    print('Please select the option number of action you want to perform:\n')
    for temp in enumerate(TEMPLATES):
        print('{}. {}'.format(*temp))
    return '\n'
```

This function solves the question:

How should the program interact with the user?

Function load_db()

Our employee data is currently stored inside a text file. Therefore, in order to access them, we need to open the file. Therefore the input that this function expects is the path to the file on the disk.

Here we see the **context manager** in use (statement `with open(path) as employees_file`). We therefore do not need to explicitly close `employees_file`, because context manager does that for us.

```
def load_db(path):
    with open(path) as employees_file:
        content = employees_file.read()
    return eval(content)
```

We could of course write the above function without the use of context manager as:

```
def load_db(path):
    employees_file = open(path)
    content = employees_file.read()
    employees_file.close()
    return eval(content)
```

The other **important thing about this function is** the use of built-in function `eval()`. This function expects an input in form of a string and then it converts this string into Python code.

This function solves the question:

Osnova

Where to get the employee data from?

Function load_template()

Load template also uses help of context manager. We also apply string formatting here. The only task of this function is to read and return the contents of the selected template.

```
1  def load_template(template_name):
2      template_path = '{}/{}.txt'.format(TEMPLATE_PATH,template_name)
3
4      with open(template_path) as template_file:
5          template = template_file.read()
6
7      return template
```

Function write_file()

Write file function has to open a new blank file, therefore we can use the `'w'` mode inside `open()` function.

We insert all the data that the template expects using dictionary unpacking and then we write the resulting string into the new file:

`new_file.write(template.format(**employee))`

```
1  def write_file(employee,template, template_name):
2      filename =
   '{}_{}.txt'.format(template_name,employee['full_name'])
3      filepath = '{}/{}'.format(CONTRACTS_PATH, filename)
4
5      with open(filepath,'w') as new_file:
6          new_file.write(template.format(**employee))
```

Running the script

100% z Lekce 9

Osnova

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       |       2,693,880       |
|       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       |
......
```

100% z Lekce 9

1. The amount of money lent = Loan

Osnova

2. Number of years in which the mortgagte 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:

```
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 = 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

```
total_amount = monthly_payment * number_of_months
```

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

```
total_interest = total_amount - loan
```

4. The listing of individual monthly payments decomposed into

- **payment number** (from 1 until the last month's number)
- **interest part** = the amount the bank is earning for lending us money.

It has to be calculated for every single month as follows:

```
monthly_interest = L * r
```

where:

100% z Lekce 9

is the monthly interest rate converted from the yearly interest rate that bank offers us (e.g. 3% / 12 = 0.25)

- **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 a 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
|
|       4         |    4,653    |       2,830        |        2,671,431
|
|       5         |    4,640    |       2,843        |        2,663,948
|
|       6         |    4,627    |       2,856        |        2,656,465
|
```

Osnova

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

Function main()

Our program runs as function `main()`. This function first collects user inputs, then, it calculates the necessary numbers and finally it generates a formatted table and writes it into a file called `mortgage.txt`.

```
def main():

    loan,interest_rate,num_periods = collect_inputs()

    monthly_rate = interest_rate/100/12
    pmt = monthly_payment(loan,monthly_rate,num_periods)
    total_to_pay = total_mortgage(pmt,num_periods)
    interest = total_interest(total_to_pay, loan)
    pmts = payments(pmt,total_to_pay,monthly_rate,num_periods)


    header_inputs=(loan,num_periods,interest,pmt)
    table = make_table(pmts, header_inputs)

    print(table)
```

```
18          file.write(table)
```

Osnova

Function collect_inputs()

This function returns a tuple of 3 values, that are necessary to calculate monthly mortgage payments.

```
1  def collect_inputs():
2
3      loan = float(input('How much do you want to borrow?: '))
4      interest_rate = float(input('At what rate?: '))
5      num_periods = int(input('How many years?: ')) * 12
6      print()
7
8      return loan, interest_rate, num_periods
```

Function monthly_payment()

This function calculates only the amount of money, we will have to send to our bank. It contains a single formula included also in the function comment.

```
1  def monthly_payment(loan,monthly_rate,num_periods):
2      '''calculate the monthly payment:
3
4      monthly_payment = L*(r * (1 + r)**n) / ((1 + r)**n-1)
5      '''
6
7      return round((loan
8                    * (monthly_rate * (1+monthly_rate)**num_periods)
9                    / ((1+monthly_rate)**num_periods - 1)))
```

Function total_mortgage()

Here we calculate, how much money we will pay in those X years to our bank. Number of periods is reference to the number of months, we will pay our mortgage.

100% z Lekce 9

```
    monthly_payment * num_periods
```

Osnova

Function total_interest()

This is the amount of money the bank will earn from us.

```
def total_interest(mortgage,loan):
    '''calculate the total interest'''

    return mortgage - loan
```

Function payments()

Here we already calculate payment for each and every month during our mortgage. This will serve as an input for the table formatting machinery we will see next.

We collect values for each month into a tuple, which we then append to the payments list.

At the end the function returns the generated list of payments.

```
def payments(pmt,total_to_pay, monthly_rate,num_periods):
    '''generate records for all monthly payments in form:

    |  Payment    |  Interest |    Principal     |      Left to Pay
|
    '''

    payments = []

    for month in range(1, num_periods+1):

        month_interest = round(total_to_pay * monthly_rate)

        payments.append((month,month_interest,

                         round(pmt - month_interest),
                         round(total_to_pay)))
```

100% z Lekce 9

```
18
    Osnova
20      return payments
```

Formatting Solution

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

Click to see our solution

Function make_table()

The main formatting function is `make_table()` . This function creates a formatted table in form of a string, with each row on a new line. Therefore the expression that follows the return keyword: `'\n'.join(formatted_table)`

The task of this function is to:

1. generate a header
2. merge the header with the input data (payments): `table = header + pmts` - `+` operation on lists merges them into one list
3. prepare templates for the header and the data part of the table
4. generate the formatted table
5. we have decided to insert separation lines between the header rows, separating them from the data as well

```
1  def make_table(pmts, header_inputs):
2
3      header = make_header(*header_inputs)
4      table = header + pmts
5      widths = column_widths(table)
```

100% z Lekce 9

```
    Osnova    ed_table = format_rows(header, header_template,widths,[])
11      formatted_table = format_rows(pmts, data_template,
    widths,formatted_table)
12
13      formatted_table = insert_lines(formatted_table, (1,3))
14
15      return '\n'.join(formatted_table)
```

Function make_header()

First helper function used inside the `make_table()` function. This function just returns the values, that should be later inserted into the header template. We have here two rows inside the header being returned.

```
1  def make_header(loan,num_periods,interest,pmt):
2
3      return [[ 'Loan: %d'% loan,'Years: %d'% (num_periods/12),
4              'Interest: %d' % interest,'Monthly Payment: %d' %pmt],
5              ['Payment','Interest', 'Principal','Left to Pay']]
```

Function column_widths()

Second helper function used inside the `make_table()` function. This function provides necessary information for the `format_rows()` function, which needs it in order it can format each column with proper width.

We use here a small trick in this function: `col_width = len(max(column, key=len)) + 4`

This trick makes use of function `max()`, which allows us to specify, what we want to compare, when we determine the maximum value in the sequence (`column`). We specify this by the parameter `key` . In this specific case, we say - compare all the items based on their length (using the `len` function). Then `max()` return the item, that is the longest among all in the sequence. Our goal is to determine its length, therefore we enclose the `max()` inside `len()` function. We add value 4 to the result to make sure there are 2 spaces on each side of the value.

```
    num_cols = len(table[0])
    col_widths = {}

    for col_num in range(num_cols):

        column = extract_column(col_num, table)

        col_width = len(max(column, key=len)) + 4
        col_widths['w%d'%(col_num+1)] = col_width

    return col_widths
```

Function extract_column()

This is a helper function for `column_widths()` in order to avoid nesting for loops and finishing with unreadable code. This function has the only task of extracting a given columns from the table.

```
def extract_column(col_num, table):
    column = []
    for row in table:
        column.append(str(row[col_num]))
    return column
```

Function format_rows()

This is the third helper functions inside `make_table()`. We use different formatting template for headers and the data and therefore we have extracted this functionality into a separate function. All it does is to collect strings representing formatted rows into a list.

The first call to this function in `make_table()` just passes an empty list for `formatted_table`. The second call passes the result of the previous call to this parameter.

```
def format_rows(data, template,widths,formatted_table):

    for row in data:
```

```
    return formatted_table
```

Osnova

Function insert_lines()

This is the last helper function inside `make_table()`. All it does is to insert a line of characters between the two header lines and then separation between headers and the data.

This function needs to know, the width of the whole table and not only of one column. As the table is now 1-dimensional list of strings, it is ok, to use the trick with `max()` function and the `key` parameter set to function `len`.

We can flexibly decide, where we want those separations to be inserted by passing a sequence of integers to the `row_nums` parameter.

```
1  def insert_lines(table, row_nums, sign='='):
2
3      width = len(max(table, key=len))
4      for row_num in row_nums:
5          table.insert(row_num,sign*width)
6
7      return table
```

That means we could create a table, where all the rows are separated by one line:

```
|  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
|
```

100% z Lekce 9

Osnova

```
==========================================================================
|       5        |    4,640    |        2,843        |        2,663,948
|
==========================================================================
```

DALŠÍ LEKCE