General

If Python needs to read the next line from a text file, it will continue reading until the first newline character (`'\n'`) or the end of the file is reached. The last character of the line that was read from the file will therefore usually be a newline (unless the last line of the file did not end in a newline).

The string method `rstrip` can be used to remove the trailing newline at the end of a line. In case this method is called without any arguments, all whitespace characters (spaces, tabs and newlines) at the end of the line will be removed. To make sure that only the newline is removed from the end of the line, you may pass the newline character as an argument to the method `rstrip`.

```
>>> line = infile.readline()
>>> line
'This is the next line in the file.\n'
>>> line.rstrip('\n')
'This is the next line in the file.'
```

Copy text file to Eclipse

If you want to locally test your solution for an assignment using text files, you must also make sure to have a local copy of the text files. Otherwise the test cases of the doctest will not be able to access these text files. The text files that are used in a given doctest are always linked in the description on top of the doctest. You can inspect the content of these text files in your browser by clicking this link.

The most general procedure to obtain a local copy of these text files in Eclipse goes as follows:

- open the text file in your browser
- copy the file content to the pastebin (`CTRL-A` + `CTRL-C`)
- create a new text file in Eclipse
  - right click the directory that needs to contain the text file (you must make sure that the text file is in the same directory as your Python script)
  - chose the menu item `New` and then the menu item `File`
  - enter the correct name of the file under `File name`; make sure that the file extension must also be given (usually `.txt`)
- paste the content of the pastebin into the file (`CTRL-V`)

The following screenshot show you the way.

If you submit a solution to Pythia, the platform will make sure that the necessary text files are in the same directory of the Python script.

Rorschachtest

General information

Remarks

The built-in function `print` can be used to write the string representation of a result to a file. This can be done by making use of the optional parameter `file` of the function.

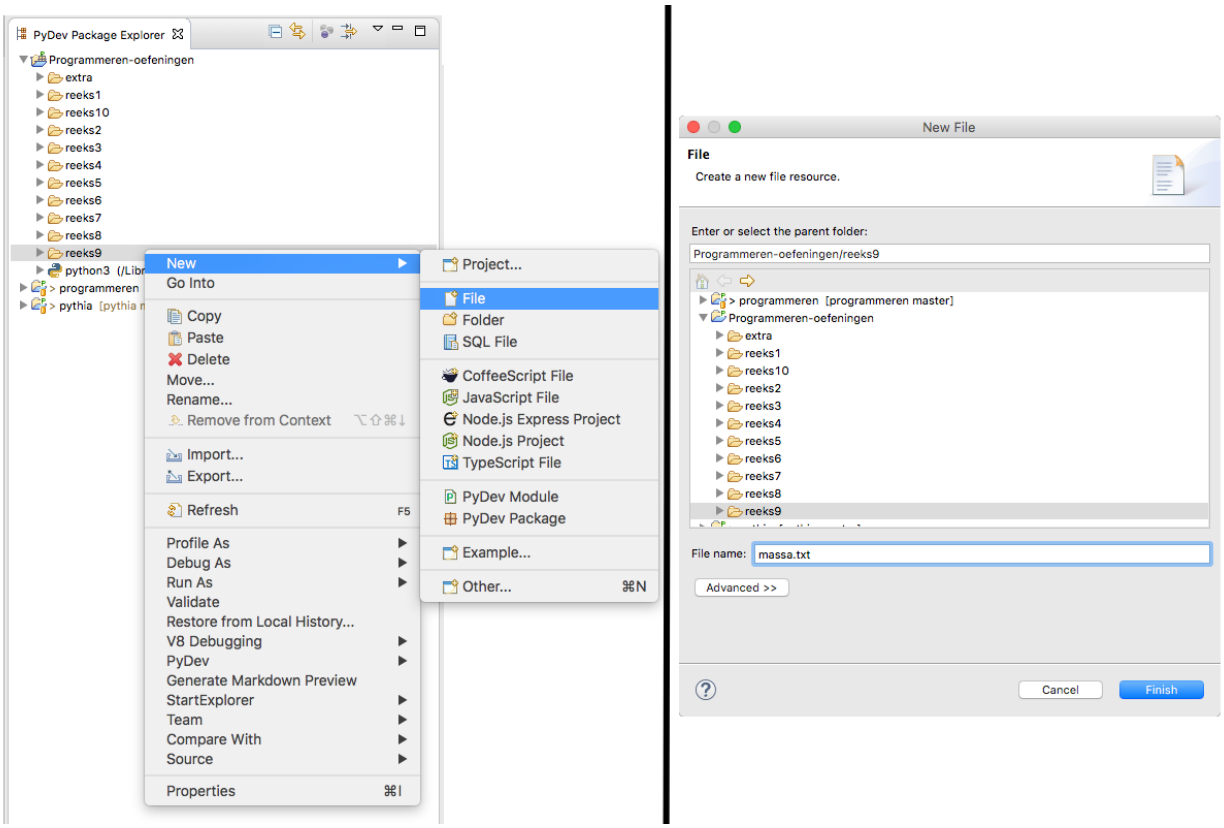

Figure 1: menu new file

By default, the function `print` will write the result to the special file `sys.stdout` (the default value of the parameter `file`) that for example might be attached to the Console window of Eclipse. The same effect can be obtained by passing the value `None` to the parameter `file`.

By passing a file object that was opened for writing to the parameter `file` of the function `print`, the string representation of the result is written to this file.

```
>>> line1 = 'This is the first line.'
>>> line2 = 'This is the second line.'
>>> print(line1)
This is the first line.
>>> print(line2, file=None)
This is the second line.

>>> outfile = open('output.txt', 'w')
>>> print(line1, file=outfile)
>>> print(line2, file=outfile)
>>> outfile.close()
>>> infile = open('input.txt', 'r')
>>> infile.readline()
'This is the first line.\n'
>>> infile.readline()
'This is the second line.\n'
>>> infile.readline()
''
```

Isomorse

General information

Remarks

The character that represents a tab in the ASCII table, may be represented in a Python string as `'\t'`. The string method `split` can be used to split a string into a list of substrings. In case no argument is passed to the method, the method will remove all leading and trailing whitespace characters (spaces, tabs and newlines), and then split the string into the substrings that are separated from each other by one or more whitespace characters.

In case a string argument is passed to the string method `split`, the method will use this argument to split the string at each occurrence of the argument. Say, for example, that the string contains two consecutive spaces, then the string method `split` without an argument will split once at that position, whereas the string method `split` with a space character as an argument will split twice at that position. In the latter case, an empty string will result as the substring between the two spaces.

```
>>> text = 'a;b  c;d;e\t  f'
>>> text.split()
['a;b', 'c;d;e', 'f']
>>> text.split(' ')
['a;b', '', 'c;d;e\t', '', 'f']
>>> text.split(';')
['a', 'b  c', 'd', '', 'e\t  f']
>>> text.split('\t')
['a;b  c;d;e', '  f']
```

Specific information

The function `pattern` has two optional parameters `complement` and `mirror` that take Boolean values (default value: `False`). Depending on the values passed to these optional parameters, the function might have to do up to three tasks:

- convert the given word into its morse code pattern
- determine the complement of the morse code pattern
- determine the inverse of the morse code pattern

Although it is possible to directly generate the requested pattern, this unnecessarily complicates matters. In this case, it is far better to apply the *divide and conquer strategy* by applying each of the tasks in succession. This means that you can first convert the given word into morse code, and only then compute the complement and/or inverse of the morse code pattern is requested.

Nonogram

Specific information

The implementation of the function `nonogram` needs to convert a line (a string) containing a description of a row in the nonograp puzzle into a list of tuples of two integers. This can be done following the strategy that is outlined in the following schematic.

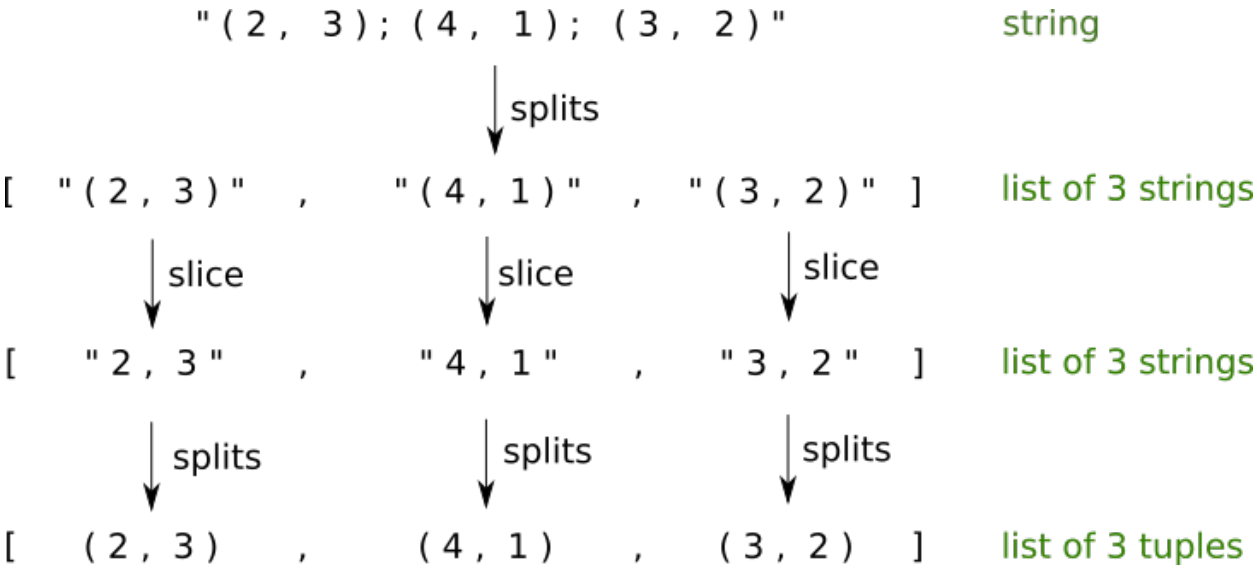

Figure 2: process description

In addition, while implementing the function `nonogram` you must pay attention to the fact that the string representation of a row of the solution can only be generated if the width of the solution is known. As a result, the descriptions of the rows of the nonogram puzzle must be processed twice. A first time to determine the maximal width of the rows and a second time to generate the rows of the solution based on the maximal width.

There are two ways of doing this. The first option is to iterate over the line in the solution file twice. In doing so, don't forget to close the file after the first iteration, and then opening it again, so that the second iteration again starts at the first line of the file. You can also make use of the file method `seek` to move the

file pointer back to the start of the file, without the need to open the file twice. The second solution is to store the lines in a list while processing them a first time. Afterwards, the file can be closed an the lines can be processed a second time by traversing the lines in the list.

Close neighbours

General information

Remarks

The `math` module from the Python Standard Library defines a couple of **trigonometric functions** such as the sine function (`sin`), the cosine function (`cos`) and the tangent function (`tan`). It's important to pay attention to the fact that these functions expect an angle expressed in radians, and not in degreees. Luckily enough, the `math` module also defines functions to convert an angle expressed in degrees into radians (`radians`) and *vice versa* (`degrees`).

```
>>> import math
>>> angle = 90
>>> radians = math.radians(angle)
>>> radians
1.5707963267948966
>>> radians == math.pi / 2
True
>>> math.cos(radians) # must evaluate to 0, but note the rounding error
6.123233995736766e-17
>>> math.sin(radians)
1.0
```

The `math` module from the Python Standard Library defines a couple of **cyclometric functions** (or inverse trigonometric functions) such as the arcsine function (`asin`), the arccosine function (`acos`) and the arctangent function (`atan`). The domain of the arcsine and the arccosine functions is $[-1, 1]$. This means that these function only take *floating point* values from the interval $[-1, 1]$. As a result, you must take care that rouding errors do not cause values outside this domein. The following example illustrates how the detrimental effect of rounding errors can be remedied.

```
>>> import math
>>> value = 1.00001
>>> math.acos(value)  # compute arccosine
Traceback (most recent call last):
ValueError: math domain error
>>> value = max(-1.0, min(value, 1.0)) # guarantee that value is in interval [-1, 1]
>>> value
1.0
>>> math.acos(value)
0.0
```