

Python

A quickstart into the key concepts of programming
IO



IO

io.ipynb



Getting data inside your program

- Hard coding
- Console IO
- File IO



Console Input Output

- Standard file handles:
 - `sys.stdin`: standard input (keyboard, pipe in)
 - `sys.stdout`: standard output (screen, pipe out)
 - `sys.stderr`: standard error (screen, pipe out)
- Input: read from the console
 - `input([prompt])` returns a value (string)
 - To convert the string to a number, cast the string to the desired data type with `int()`, `float()`
- *File: input_1.py*



Console Output

- The `print` function is the most commonly used command to print information to the “standard output device” which is normally the screen.
- 2 modes to use `print`.
 - Simple print
 - The easiest way to use the `print` command is to list the variables to be printed, separated by comma.
 - Formatted print
 - The more sophisticated way of formatting output uses a syntax very similar to Matlab's `fprintf` (also similar to C's `printf`).



Print

- `print()` writes objects to `sys.stdout`, adds `'\n'` (or `'\r\n'`) and applies `str()` conversion function by default
- `print` automatically puts a single space between items and a newline at the end.
- **Syntax:** `print(objects, sep=' ', end='\n', file=sys.stdout, flush=False)`
- **Parameters**
 - `objects` - objects to be printed.
 - `sep` - objects are separated by `sep`. Default value: `' '`
 - `end` - end is printed at last
 - `file` - must be an object with `write(string)` method. Default: `sys.stdout` (screen)
 - `flush` - If `True`, the stream is forcibly flushed. Default value: `False`



Format output

- **fill-in-the-braces**

- string type has `format()` method which takes arguments and returns another string which is the original with the method's arguments inserted into it in certain places.
- `format()` replaces each pair of curly brackets with the corresponding argument.

- **Syntax:** `S.format(item0, item1, item2, ..., itemk)`

- S: string consisting of substrings and formatting instructions for each item in the form of `{index:format-specifier}`.
- index refers to the item and format-specifier specifies how the item is formatted. (Optional)



Format output

These are indicated by a pair of curly braces `{}`

- Empty braces

`'The {} in the {}'.format('car', 'room')` returns 'The car in the room'

- Braces with index

`'{2} color {1}, {0} {3}'.format('dark', 'brown', 'yellow', 'red')` returns 'yellow color brown, dark red'

- Braces with keywords

- braces contain a keyword and parameter of the form `<keyword>=<value>`
`'{c} color {b}, {a} {d}'.format(a='dark', b='brown', c='yellow', d='red')` returns 'yellow color brown, dark red'



Format output

```
count = 5
amount = 45.56
print('count is {0} and amount is
{1:9.6f}'.format(count, amount))
```



Format specifiers

Conversion	Meaning
d	Signed integer decimal.
i	Signed integer decimal.
o	Unsigned octal.
x	Unsigned hexadecimal (lowercase).
X	Unsigned hexadecimal (uppercase).
e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase).
f	Floating point decimal format.
F	Floating point decimal format.
g	Same as "e" if exponent is greater than -4 or less than precision, "f" otherwise.
G	Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise.
c	Single character (accepts integer or single character string).
r	String (converts any python object using repr()).
s	String (converts any python object using str()).
%	No argument is converted, results in a "%" character in the result.

https://www.python-course.eu/python3_formatted_output.php



Format specifiers

- Justification codes
 - < left justifies
 - > right justifies
 - ^centers
- Filling characters
 - Character after : pads with that character
`\{: *^9\}'.format(3)` returns `'***3***'`
 - + before the field width forces a preceding sign
- Character functions
 - `ord` - Syntax: `ord(< char >)`
 - `chr` - Syntax: `chr(< int >)`



Format specifiers

- single quote \'
- double quote \"
- backslash \\
- alert (bell) \a
- backspace \b
- formfeed \f
- newline \n
- return \r
- tab \t
- vertical tab \v
- <https://realpython.com/python-input-output/>



Command line arguments

- All of the command line parameters live inside of `sys.argv`, so `import sys` is needed.
- `sys.argv` is a list in Python, which contains the command-line arguments passed to the script.
- The name of the program is always the first item (`argv[0]`)
- The elements of `argv` are strings
- *File: `command_line_io.py`*
- <https://realpython.com/python-command-line-arguments/>



Command line arguments

File: `command_line_io.py`

```
import sys
print('This is the name of the
script: ', sys.argv[0])
print('Number of arguments: ',
len(sys.argv))
print('The arguments are: ',
sys.argv)

for arg in sys.argv:
    print(type(arg), arg)
```

Command line

```
(base) PS C:\temp\Develop\PythonDev>
python .\command_line_io.py help me
This is the name of the script:
.\command_line_io.py
Number of arguments: 3
The arguments are:
['.\command_line_io.py', 'help',
'me']
<class 'str'> .\command_line_io.py
<class 'str'> help
<class 'str'> me
(base) PS C:\temp\Develop\PythonDev>
```



Specific information on file io

- https://python.camden.rutgers.edu/python_resources/python3_book/files.html
- <http://www.pythonforbeginners.com/files/reading-and-writing-files-in-python>
- <https://www.datacamp.com/community/tutorials/reading-writing-files-python>



File IO

- See: <https://realpython.com/read-write-files-python/>
- File IO is handled natively in the language.
- Working with files is a lot like working with a notebook.
 - To use a notebook, it has to be opened.
 - When done, it has to be closed.
 - While the notebook is open
 - it can either be read from or written to.
 - in its natural order or skip around.
- Write/Read text files in Python is straightforward. At the simplest level, it involves these steps:
 - Opening the file and associating it with a file handle.
 - Using the file handle: read from or write to a file
 - Closing the file handle to commit to a disk file. (Do not forget!)



File IO: access mode

r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

https://www.tutorialspoint.com/python/python_files_io.htm



File IO: access mode

wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

https://www.tutorialspoint.com/python/python_files_io.htm



Write to a file

- `open()`: get a file object handle
 - File objects contain methods and attributes that can be used to collect information about the file.
 - Syntax: `file_object = open('filename', 'mode', 'encoding')`

- A simple example

```
outfile = open("testwrite.txt", "w")
outfile.write("Line number 1\n")
outfile.close()
```

- *File: file_io_write_1.py*



Write to a file

- **write()** is used to write a string to an already opened file
 - The **write()** function only works with **string** arguments.
 - If you want to write the results of a numerical variable to an output file, you need to convert the number into a string. This is done using the **str()** function
- **writelines()** expects an iterable as argument (an iterable object can be a tuple, a list, a string, or an iterator in the most general sense). Each item contained in the iterator is expected to be a string.

Write to a file

File: *file_io_write_1.py*

```
fp = open('testfile-write.txt','w')
fp.write('Hello World')
fp.write('Hello World \n')
# write the list
# note, here print is used
# It inserts spaces between arguments and
# appends the line terminator.
l = [0, 1, 2, 3]
print(l, file=fp)
for i in l:
    print(i, file=fp)
for i in l:
    print(i, end=' ', file=fp)
# write the individual elements
for i in l:
    fp.write(str(i))
# write the individual elements with a
# separator
for i in l:
    fp.write(str(i) + '/')
fp.write('\n')
fp.close()
```

testfile-write.txt

```
Hello WorldHello World
[0, 1, 2, 3]
0
1
2
3
0 1 2 3 01230/1/2/3/
```



Reading a file

- Text files are sequences of lines of text, we can use the **for** loop to iterate through the file object (a built-in Python functionality)
- A line of a file is defined to be a sequence of characters up to and including the newline character (`\n`).
- Pattern for processing each line of a text file:

```
for line in myFile:
    statement1
    statement2
    ...
```

- File: *file_io_forLoop.py*



Reading a file

File: file_io_forLoop.py

```
txtfile =  
open("test_text.txt", "r")  
  
for aline in txtfile:  
    values = aline.split()  
    if len(values) > 10:  
        print('some text',  
values[0], values[1], ' and  
also ', values[10] )  
  
txtfile.close()
```

result

```
some text Lorem ipsum and also quam  
some text Mauris venenatis and also Cras  
some text Nunc dictum, and also vel  
some text Aenean accumsan and also varius  
some text Curabitur commodo and also id  
some text In faucibus and also diam.  
some text Orci varius and also ridiculus  
some text Curabitur interdum, and also nulla,  
some text Aliquam vitae and also nibh  
some text Phasellus posuere and also mollis  
some text Sed malesuada and also sodales  
some text Nulla condimentum and also vel  
some text Cras et and also nisi  
some text Vestibulum vel and also ut  
some text Suspendisse id and also tincidunt  
some text Nulla sed and also urna  
some text Donec tristique and also Donec
```



Reading a file

Read from a file handle that has been connected to a file that is open for reading.

- `file.readline()` : read a file line by line
 - The `readline()` function reads a single line from the file and advances the line pointer by one line.
 - It is possible to call `readline()` over and over until the whole file is read in, one line at a time.
 - Usually the `readline()` function is used when you only want to read a single line from the file, i.e. header line.
 - If there is an argument, it indicates the number of bytes to be read.
 - File: `file_io_readline.py`
- `file.readlines()` : return every line in the file, properly separated
 - Reads all the lines of file into a list, with each element string consisting of the line (with `\n`).
 - File: `file_io_readlines.py`
- `file.read()` : reads in the entire contents of the file as a single string, used when parsing JSON files, HTML or XML files
- `file.read(n)` : reads up to n chars as a single string



Reading a file

- Read all characters in the file

```
fp = open ("filename", "r")
```

```
fp.read()
```

Specify number of bytes

```
fp = open ("filename", "r")
```

```
fp.read(n)
```

- Python reads only strings. You must parse the line or lines yourself.
- Useful functions
 - `string.strip(s)`
 - `string.split(s)`
- File: *file_io_3.py*
- File: *file_io_readline.py*
- File: *file_io_readlines.py*



File IO

- The Pythonic way:

```
with open('file_path', 'w') as file:
```

```
    file.write('hello world !')
```

- The `with` statement automatically takes care of closing the file once it leaves the `with` block, even in cases of error.
 - no need to call `file.close()` when using `with` statement.
 - `with` statement itself ensures proper acquisition and release of resources.



I/O and data formats

- <https://github.com/gjbex/training-material/tree/master/Python/DataFormats>



Libraries & data formats

- Problem: Not portable code, data type size? Encoding? little /big endian?
- Use the *batteries* that are included!
- Standard library (Python 3.x)
 - Comma separated value files: `csv`
 - Configuration files: `ConfigParser`
 - Semi-structured data: `json`, `htmllib`, `sgmlib`, `xml`
- Non-standard libraries
 - Images: `scikit-image`
 - HDF5: `pytables`
 - `pandas`
 - Bioinformatics: `Biopython`
- <https://github.com/gjbex/training-material/tree/master/Python>



Reading and writing csv

- <https://realpython.com/python-csv/>
- <https://www.alexkras.com/how-to-read-csv-file-in-python/>
- Structure of a csv file
 - column 1 name, column 2 name, ...
 - first row data 1, first row data 2, ...
 - second row data 1, second row data 2, ...
 - ...
- csv library provides functionality to both read from and write to csv files.



Reading csv

- Steps
 - Open the file
 - Create a CSV reader
 - Skip first line (header)
 - For every line (row) in the file, do something
- *File: file_read_csv_list.py*



Writing csv

- **writer()**
 - function from csv module returns a writer object that converts data into a delimited string and stores in a file object. The function needs a file object with write permission as a parameter.
 - Every row written in the file issues a newline character. To prevent additional space between lines, newline parameter is set to "".
- The writer class has following methods
 - **writerow()**
 - This function writes items in an iterable (list, tuple or string), separated by comma
 - **writerows()**
 - This function takes a list of iterables as parameter and writes each item as a comma separated line of items in the file.
- <https://www.tutorialspoint.com/reading-and-writing-csv-file-using-python>
- *File: file_write_csv.py*

