Basic Input and Output

Examples

Using the print function

In Python 3, print functionality is in the form of a function:

```
print("This string will be displayed in the output")
# This string will be displayed in the output

print("You can print \n escape characters too.")
# You can print escape characters too.
```

In Python 2, print was originally a statement, as shown below.

```
print "This string will be displayed in the output"
# This string will be displayed in the output

print "You can print \n escape characters too."
# You can print escape characters too.
```

Note: using from __future__ import print_function in Python 2 will allow users to use the print() function the same as Python 3 code. This is only available in Python 2.6 and above.

Function to prompt user for a number

```
def input_number(msg, err_msg=None):
    while True:
        try:
            return float(raw_input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)


def input_number(msg, err_msg=None):
    while True:
        try:
            return float(input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)
```

And to use it:

```
user_number = input_number("input a number: ", "that's not a number!")
```

Or, if you do not want an "error message":

```
user_number = input_number("input a number: ")
```

Input from a File

Input can also be read from files. Files can be opened using the built-in function open . Using a 🔲 with <command> as <name> syntax (called a 'Context Manager') makes using open and getting a handle for the file super easy:

```
with open('somefile.txt', 'r') as fileobj:
    # write code here using fileobj
```

This ensures that when code execution leaves the block the file is automatically closed.

Files can be opened in different modes. In the above example the file is opened as read-only. To open an existing file for reading only use r . If you want to read that file as bytes use rb . To append data to an existing file use a . Use w to create a file or overwrite any existing files of the same name. You can use r+ to

open a file for both reading and writing. The first argument of open() is the filename, the second is the mode. If mode is left blank, it will default to r .

```
# let's create an example file:
with open('shoppinglist.txt', 'w') as fileobj:
    fileobj.write('tomato\npasta\ngarlic')

with open('shoppinglist.txt', 'r') as fileobj:
    # this method makes a list where each line
    # of the file is an element in the list
    lines = fileobj.readlines()

print(lines)
# ['tomato\n', 'pasta\n', 'garlic']

with open('shoppinglist.txt', 'r') as fileobj:
    # here we read the whole content into one string:
    content = fileobj.read()
    # get a list of lines, just like int the previous example:
    lines = content.split('\n')

print(lines)
# ['tomato', 'pasta', 'garlic']
```

If the size of the file is tiny, it is safe to read the whole file contents into memory. If the file is very large it is often better to read line-by-line or by chunks, and process the input in the same loop. To do that:

```
with open('shoppinglist.txt', 'r') as fileobj:
    # this method reads line by line:
    lines = []
    for line in fileobj:
        lines.append(line.strip())
```

When reading files, be aware of the operating system-specific line-break characters. Although for line in fileobj automatically strips them off, it is always safe to call strip() on the lines read, as it is shown above.

Opened files (fileobj in the above examples) always point to a specific location in the file. When they are first opened the file handle points to the very beginning of the file, which is the position 0 . The file handle can display it's current position with tell :

```
fileobj = open('shoppinglist.txt', 'r')
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 0.
```

Upon reading all the content, the file handler's position will be pointed at the end of the file:

```
content = fileobj.read()
end = fileobj.tell()
print('This file was %u characters long.' % end)
# This file was 22 characters long.
fileobj.close()
```

The file handler position can be set to whatever is needed:

```
fileobj = open('shoppinglist.txt', 'r')
fileobj.seek(7)
pos = fileobj.tell()
print('We are at character #%u.' % pos)
```

You can also read any length from the file content during a given call. To do this pass an argument for read() . When read() is called with no argument it will read until the end of the file. If you pass an argument it will read that number of bytes or characters, depending on the mode (rb and r respectively):

```
# reads the next 4 characters
# starting at the current position
next4 = fileobj.read(4)
# what we got?
print(next4) # 'cucu'
# where we are now?
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 11, as we was at 7, and read 4 chars.

fileobj.close()
```

To demonstrate the difference between characters and bytes:

```
with open('shoppinglist.txt', 'r') as fileobj:
    print(type(fileobj.read())) # <class 'str'>

with open('shoppinglist.txt', 'rb') as fileobj:
    print(type(fileobj.read())) # <class 'bytes'>
```

Printing a string without a newline at the end

In Python 2.x, to continue a line with print , end the print statement with a comma. It will automatically add a space.

```
print "Hello,",
print "World!"
# Hello, World!
```

In Python 3.x, the print function has an optional end parameter that is what it prints at the end of the given string. By default it's a newline character, so equivalent to this:

```
print("Hello, ", end="\n")
print("World!")
# Hello,
# World!
```

But you could pass in other strings

```
print("Hello, ", end="")
print("World!")
# Hello, World!

print("Hello, ", end="<br>")
print("World!")
# Hello, <br>World!

print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!
```

If you want more control over the output, you can use sys.stdout.write :

```
import sys

sys.stdout.write("Hello, ")
sys.stdout.write("World!")
# Hello, World!
```

Read from stdin

Python programs can read from unix pipelines . Here is a simple example how to read from stdin :

```
import sys

for line in sys.stdin:
    print(line)
```

Be aware that sys.stdin is a stream. It means that the for-loop will only terminate when the stream has ended.

You can now pipe the output of another program into your python program as follows:

```
$ cat myfile | python myprogram.py
```

In this example cat myfile can be any unix command that outputs to stdout .

Alternatively, using the fileinput module can come in handy:

```
import fileinput
for line in fileinput.input():
    process(line)
```

Using input() and raw_input()

raw_input will wait for the user to enter text and then return the result as a string.

```
foo = raw_input("Put a message here that asks the user for input")
```

In the above example foo will store whatever input the user provides.

input will wait for the user to enter text and then return the result as a string.

```
foo = input("Put a message here that asks the user for input")
```

In the above example foo will store whatever input the user provides.

Syntax

Parameters

Remarks