Week 10: Input and Output

1. Writing Output to the Console

1.1 Revisiting the `print()` function

Let's come back to `print()` function and discover more about its parameters

In [51]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
```

In [52]:
```
a, b = 1, 2
```

In [53]:
```
print(a, b)
```

```
1 2
```

In [54]:
```
print(a, b, sep=',')
```

```
1,2
```

In [55]:
```
print(a, end='.\n')
print(b, end='.\n')
```

```
1.
2.
```

In [56]:
```
for i in range(5):
    print(i, end=', ')
```

```
0, 1, 2, 3, 4,
```

1.2 Using the `format()` method of `String`

Basic usage of the `str.format()` method looks like this:

In [57]:
```
print("first {} second {}".format(a, b))
```

```
first 1 second 2
```

In [58]:
```
print("first {0} second {1}".format(a, b))
```

```
first 1 second 2
```

In [59]:
```
print("first {1} second {0}".format(a, b))
```

```
first 2 second 1
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the str.format() method. A number in the brackets can be used to refer to the position of the object passed into the str.format() method.

If keyword arguments are used in the str.format() method, their values are referred to by using the name of the argument.

---

2. Reading and writing files

2.1 Create or Open with open()

You need to call the `open()` function before you do the following:

- Read an existing file
- Write to a new file
- Append to an existing file
- Overwrite an existing file

`open()` returns a *file object*, and is most commonly used with two positional arguments, although it has many more arguments with default values:

```
fileobj = open(filename, mode)
```
Here's a brief explanation of the pieces of this call:

- `fileobj` is the file object returned by open(). It is used to read and write to a file.
- `filename` is the string name of the file or path to a file.
- `mode` is a string indicating the file's type and what you want to do with it. It is a string of up to two characters.
  - The first letter of mode indicates the operation:
    - 'r' when the file will only be read,
    - 'w' for only writing (an existing file with the same name will be erased),
    - 'a' opens the file for appending; any data written to the file is automatically added to the end.
    - 'r+' opens the file for both reading and writing.
  - The second letter of mode is the file's type:
    - `t` (or nothing) means text.
    - `b` means binary. Actually, mode argument is optional; `'r'` will be assumed if it's omitted.

In [60]:
```
f = open("temp.txt", 'w')
```

Last, you need to close the file to ensure that any writes complete, and that memory is freed.

In [61]:
```
f.close()
```

2.2 Write a Text File with print()

Our `print()` function can also be used to write to a file. If we call `help` on `print`, we can see that it has a keyword argument called `file` which has default value of `None`. Therefore, if we don't specify anything for `file`, `print` writes to `sys.stdout` which is read as `system's standard output` that is our `terminal` if we simply speak. However, if we supply `fileobj` of some open file, then we can write to this file.

In [62]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
```

In [63]:
```
f = open("temp.txt", 'w')
for i in range(1, 5):
    print("This is a line", file=f)
f.close()
```

In [64]:
```
f = open("temp.txt", 'w')
for i in range(1, 5):
    print("This is line", i, sep=':', end='.\n', file=f)
f.close()
```

Actually, we can use good old `print()` in a way that we used to

2.3 Write a Text File with write()

For our multiline data source, let's prepare the following

The `write` function returns the number of bytes written. It does not add any spaces or newlines, as `print` does. As before, you can also `print` a multiline string to a text file:

In [65]:
```
f = open("temp.txt", 'a')
f.write("This is line 5!")
f.close()
```

So, should you use write or print? As you've seen, by default `print` adds a space after each argument and a newline at the end. To make print work like write, pass it the following two arguments:

- `sep` (separator, which defaults to a space, ' ')
- `end` (end string, which defaults to a newline, '\n')

2.4 Read a Text File with read(), readline(), or readlines()

You can call `read()` with no arguments to read the entire file at once.

In [66]:
```
f = open("temp.txt", 'r')
help(f.read)
f.close()
```

```
Help on built-in function read:

read(size=-1, /) method of _io.TextIOWrapper instance
    Read at most size characters from stream.

    Read from underlying buffer until we have size characters or we hit EOF.
    If size is negative or omitted, read until EOF.
```

In [67]:
```
f = open("temp.txt", 'r')
f.read()
```

```
Out[67]: 'This is line:1.\nThis is line:2.\nThis is line:3.\nThis is line:4.\nThis is line 5!'
```

```
In [68]: f = open("temp.txt", 'r')
         f.read(5)
```

```
Out[68]: 'This '
```

```
In [69]: f.read(10)
```

```
Out[69]: 'is line:1.'
```

As shown in the example that follows (be careful when doing this with large files; a gigabyte file will consume a gigabyte of memory)

You can provide a maximum character count to limit how much `read()` returns at one time. Let's read 10 characters at a time and append each chunk to a poem string to rebuild the original:

```
In [70]: f = open("temp.txt", 'r')
         file_contents = ''
         while True:
             chunk = f.read(10)
             print("this is a chunk:", chunk)
             if chunk == '':
                 break
             file_contents += chunk

         print("This is file contents:", file_contents)
```

```
this is a chunk: This is li
this is a chunk: ne:1.
This
this is a chunk:  is line:2
this is a chunk: .
This is
this is a chunk: line:3.
Th
this is a chunk: is is line
this is a chunk: :4.
This i
this is a chunk: s line 5!
this is a chunk: 
This is file contents: This is line:1.
This is line:2.
This is line:3.
This is line:4.
This is line 5!
```

After you've read all the way to the end, further calls to `read()` will return an empty string (''). Detecting empty string helps us to break out of the infinite loop

You can also read the file a line at a time by using `readline()`.

```
In [71]: f = open("temp.txt", 'r')
         f.readline()
```

```
Out[71]: 'This is line:1.\n'
```

```
In [72]: f.readline()
```

```
Out[72]: 'This is line:2.\n'
```

```
In [73]: f.readlines()
```

```
Out[73]: ['This is line:3.\n', 'This is line:4.\n', 'This is line 5!']
```

In this next example, we collect each line to rebuild the original:

The easiest way to read a text file is by using an iterator. This returns one line at a time. It's similar to the previous example but with less code:

The readlines() call reads a line at a time, and returns a list of one-line strings.

2.5 Close Files Automatically Using `with` Context Manager

If you forget to close a file that you've opened, it will be closed by Python after it's no longer referenced.

Python has context managers to clean up things such as open files. You use the form

```
with expression as variable:
```

In [74]:
```
with open('temp.txt', 'r') as f:
    print(f.read(5))
    print(f.read(5))

f.read()
```

```
This 
is li
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[74], line 5
      2     print(f.read(5))
      3     print(f.read(5))
----> 5 f.read()

ValueError: I/O operation on closed file.
```

as you can see calling `f.read()` outside of `with` raises an error since it closes the file automatically.

2.6 Change Position with seek()

As you read and write, Python keeps track of where you are in the file.

- The `tell()` function returns your current offset from the beginning of the file, in bytes.
- The `seek()` function lets you jump to another byte offset in the file. This means that you don't have to read every byte in a file to read the last one; you can seek() to the last one and just read one byte.

In [75]:
```
with open('temp.txt', 'r') as f:
    print("current position:", f.tell())
    print(f.read(5))
    print("current position:", f.tell())
    print(f.read(10))
    print("current position:", f.tell())
```

```
current position: 0
This 
current position: 5
is line:1.
current position: 15
```

In [76]:
```
with open('temp.txt', 'r') as f:
    print("current position:", f.tell())
    f.seek(10)
    print("current position:", f.tell())
    f.seek(f.tell() - 5)
    print(f.read(1))
    print("current position:", f.tell())
```

```
current position: 0
current position: 10
i
current position: 6
```

To change the file object’s position, use `f.seek(offset, whence)`. The position is computed from adding `offset` to a reference point; the reference point is selected by the `whence` argument.

- A whence value of 0 measures from the beginning of the file,
- 1 uses the current file position,
- and 2 uses the end of the file as the reference point. whence can be omitted and defaults to 0, using the beginning of the file as the reference point.

Important: non-zero `whence` should be used in binary mode.

3. References

1. Bill Lubanovic, "Introducing Python," 2nd edition, O'reilly, Available online
2. The Python tutorial: 7. Input and Output, https://docs.python.org/3/tutorial/inputoutput.html