Fundamentals of Information Systems

Python Programming (for Data Science)

Master's Degree in Data Science

Gabriele Tolomei
gtolomei@math.unipd.it
University of Padua, Italy
2018/2019
October 25, 2018

Lecture 5: Functions & I/O

Functions

Motivations and Basic Syntax

- Functions are the primary and most important method of **code organization** and **reuse** in Python (and any programming language, for that matters!).
- In Python, functions are declared using the `def` keyword and returned from using the `return`.
- You may have multiple `return` statements.
- If the end of a function's body is reached without encountering a `return` statement, `None` is returned automatically.

Input Arguments: *positional* vs. *keyword*

- Each function can have some number of ***positional*** arguments and some number of ***keyword*** arguments.
- Keyword arguments are typically used to specify *default values* or optional arguments.
- The main restriction is that the keyword arguments **must** follow positional arguments (if any).

In [1]:
```
'''
Example of function definition.
my_function is the name of the function
This takes 3 input arguments:
- a and b are positional arguments
- c and d are keyword arguments
'''
def my_function(a, b, c=1.5, d=3):
    if c > 1:
        return c * (a + b) ** d
    else:
        return c / (a + b) ** d

# The function above can be called in either one of the following ways:
# 1. Without keyword arguments (i.e., default keyword argument values are used)
print(my_function(2,3))
# 2.1. With keyword arguments (without specifying keywords)
print(my_function(2,3,.8,2))
# 2.2. With keyword arguments (specifying keywords in any order, disregarding the signature)
print(my_function(2,3,d=.5,c=2))
```

```
187.5
0.032
4.47213595499958
```

Namespaces, Scope, and Local Functions

- Functions can access variables in 2 different scopes: **global** and **local**.
- An alternate and more descriptive name describing a variable scope in Python is a **namespace**.
- Any variable that is assigned within a function by default is assigned to the **local namespace**.
    - The local namespace is created when the function is called and immediately populated by the function's arguments. After the function returns, the local namespace is destroyed (with some exceptions).

Global and *Local* Variable Scope

- Python variables are **local** if not otherwise declared. The reason being is that **global** variables are generally bad practice and should be avoided.
- When you define variables inside a function definition, they are local to **this** function by default. As such:
    - Anything you do to those variables in the body of the function will have no effect on other variables outside of the function, **even if they have the same name**.
    - In other words, the function body is the **scope** of those variables, i.e., the context where names with their values are associated.

Global and *Local* Variable Scope

- All variables have the scope of the block, where they are ***declared*** and ***defined***. (*They can only be used after the point of their declaration.*)
- Just to make things clear: Variables don't have to be and can't be declared in the way they are declared in programming languages like Java or C.

  Variables in Python are implicitly declared by defining them, i.e., the first time you assign a value to a variable, this variable is declared and has automatically the data type of the object which has to be assigned to it.

In [2]:
```
# Example 1: Trying to access a variable outside the local scope
# of the function where it has been defined (and declared)
def foo():
    # We define my_list within the function's body
    # This variable is assigned to the local namespace of this function
    my_list = [36, 49, 64]
    # Print the values contained in my_list
    print("Inside foo(), my_list is {}".format(my_list))

# Print the values contained in my_list before calling foo()
# NOTE: my_list has been only defined (and declared) within foo()'s scope!
print("Before calling foo(), my_list is {}".format(my_list))
# Calling foo()
foo()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-7fde73c52660> in <module>()
     10 # Print the values contained in my_list before calling foo()
     11 # NOTE: my_list has been only defined (and declared) within foo()'s scope!
---> 12 print("Before calling foo(), my_list is {}".format(my_list))
     13 # Calling foo()
     14 foo()

NameError: name 'my_list' is not defined
```

In [3]:
```
# Example 2: Access a variable inside the local scope of a function yet
# defined (and declared) outside of it
# We define a function foo() which in its body uses the variable my_list.
# Such a variable is defined (and therefore declared) before calling foo()
# Define foo()
def foo():
    # Print the values contained in my_list
    print("Inside foo(), my_list is {}".format(my_list))

# Define my_list outside the scope of foo() and before calling foo()
my_list = [42, 73, 96]
# Print the values contained in my_list before calling foo()
print("Before calling foo(), my_list is {}".format(my_list))
# Calling foo()
foo()
# As there is no local variable my_list defined in foo()'s body,
# i.e., there is no assignment to my list,
# the value from the outside global variable my_list will be used,
# as this is the only existing binding of the name 'my_list' to a proper value.
# So, we expect the output to be the list [42, 73, 96].
```

```
Before calling foo(), my_list is [42, 73, 96]
Inside foo(), my_list is [42, 73, 96]
```

Checkpoint Quiz

What will happen, if we change the value of `my_list` inside of the function `foo()`? Will this affect the outside global variable as well?

In [4]:
```
# Example 3: Modifying the value of a variable inside the local scope of a function yet
# defined (and declared) outside of it is reflected to the global variable
def foo():
    # We modify the variable my_list within the function's body
    my_list.extend([36, 49, 64])
    # Print the values contained in my_list
    print("Inside foo(), my_list is {}".format(my_list))

# Define my_list outside the scope of foo() and before calling foo()
my_list = [42, 73, 96]
# Print the values contained in my_list before calling foo()
print("Before calling foo(), my_list is {}".format(my_list))
# Calling foo()
foo()
# Print the values contained in my_list when foo() returns
print("After calling foo(), my_list is {}".format(my_list))
```

```
Before calling foo(), my_list is [42, 73, 96]
Inside foo(), my_list is [42, 73, 96, 36, 49, 64]
After calling foo(), my_list is [42, 73, 96, 36, 49, 64]
```

In [5]:
```
# Example 4: Re-binding a variable inside the local scope of a function yet
# defined (and declared) outside of it does not affect global variable
def foo():
    # We re-define the variable my_list within the function's body
    # Actually, we are re-binding the name my_list (already used) to a different object
    my_list = [36, 49, 64]
    # Print the values contained in my_list
    print("Inside foo(), my_list is {}".format(my_list))

# Define my_list outside the scope of foo() and before calling foo()
my_list = [42, 73, 96]
# Print the values contained in my_list before calling foo()
print("Before calling foo(), my_list is {}".format(my_list))
# Calling foo()
foo()
# Print the values contained in my_list when foo() returns
print("After calling foo(), my_list is {}".format(my_list))
```

```
Before calling foo(), my_list is [42, 73, 96]
Inside foo(), my_list is [36, 49, 64]
After calling foo(), my_list is [42, 73, 96]
```

Checkpoint Quiz

What if we combine the last two examples, i.e., within the `foo()` function, we first access `my_list` with a `print()`, hoping to get the value associated with the **global namespace** (outside `foo()`'s local scope), and then assigning a new value to it? Assigning a value to it, means creating a **local** variable `my_list`. So, we would have the same name `my_list` bound both to a **global** and a **local** variable in the same scope, i.e., the body of the function.

In [6]:
```
# Example 5: Access AND try to re-bind a variable inside the local scope of a function
# yet defined (and declared) outside of it
def foo():
    # Print the values contained in my_list
    print("Inside foo(), my_list is {}".format(my_list))
    # We re-define the variable my_list within the function's body
    # Actually, we are re-binding the name my_list (already used) to a different object
    my_list = [36, 49, 64]
    # Print the values contained in my_list
    print("Inside foo(), my_list is {}".format(my_list))

# Define my_list outside the scope of foo() and before calling foo()
my_list = [42, 73, 96]
# Print the values contained in my_list before calling foo()
print("Before calling foo(), my_list is {}".format(my_list))
# Calling foo()
foo()
# Print the values contained in my_list when foo() returns
print("After calling foo(), my list is {}".format(my list))
```

```
Before calling foo(), my_list is [42, 73, 96]

---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-6-324b199fd096> in <module>()
     15 print("Before calling foo(), my_list is {}".format(my_list))
     16 # Calling foo()
---> 17 foo()
     18 # Print the values contained in my_list when foo() returns
     19 print("After calling foo(), my_list is {}".format(my_list))

<ipython-input-6-324b199fd096> in foo()
      3 def foo():
      4     # Print the values contained in my_list
----> 5     print("Inside foo(), my_list is {}".format(my_list))
      6     # We re-define the variable my_list within the function's body
      7     # Actually, we are re-binding the name my_list (already used) to a different o
bject

UnboundLocalError: local variable 'my_list' referenced before assignment
```

Observations

- A variable can't be both ***local*** and ***global*** inside of a function.
- Python assumes that `my_list` in the first `print()` statement inside `foo()`'s local scope refers to the ***local*** variable `my_list` that is defined right after and **not** to the ***global*** one defined outside.
- That is why we get the `UnboundLocalError`, as Python thinks we are trying to access a variable before its definition.
- To tell Python that we want to use the ***global*** variable, we have to explicitly state this by using the keyword `global`.

In [7]:
```
# Example 6: Access AND try to re-bind a variable inside the local scope of a function
# yet defined (and declared) outside of it by explicitly telling Python that
# we are referring to the global (outside defined) variable.
def foo():
    # Explicitly tell Python we are referring to the variable my_list
    # defined outsilde this local scope (i.e., global)
    global my_list
    # Print the values contained in (global) my_list
    print("Inside foo(), my_list is {}".format(my_list))
    # We re-define the global variable my_list within the function's body
    # Actually, we are re-binding the name my_list (already used) to a different object
    my_list = [36, 49, 64]
    # Print the values contained in my_list
    print("Inside foo(), my_list is {}".format(my_list))

# Define my_list outside the scope of foo() and before calling foo()
my_list = [42, 73, 96]
# Print the values contained in my_list before calling foo()
print("Before calling foo(), my list is {}".format(my list))
# Calling foo()
foo()
# Print the values contained in my_list when foo() returns
print("After calling foo(), my_list is {}".format(my_list))
```

```
Before calling foo(), my_list is [42, 73, 96]
Inside foo(), my_list is [42, 73, 96]
Inside foo(), my_list is [36, 49, 64]
After calling foo(), my_list is [36, 49, 64]
```

Nonlocal Variables

- Pyhton 3 introduces also `nonlocal` variables.
- Those are similar to `global` variables but they can only be used inside of nested functions.
- In other words, a `nonlocal` variable has to be defined in the enclosing function scope.

In [8]:
```
# Example 7: nonlocal variables can only be used if they are defined in one of the
# enclosing nested function
def foo():
    # Trying to access my_list as if it was global turns into an error!
    nonlocal my_list
    # Print the values contained in (nonlocal) my_list
    print("Inside foo(), my_list is {}".format(my_list))

# Define my_list outside the scope of foo() and before calling foo()
my_list = [42, 73, 96]
# Print the values contained in my_list before calling foo()
print("Before calling foo(), my_list is {}".format(my_list))
# Calling foo()
foo()
# Print the values contained in my_list when foo() returns
print("After calling foo(), my_list is {}".format(my_list))
```

```
  File "<ipython-input-8-01cd382299be>", line 5
    nonlocal my_list
    ^
SyntaxError: no binding for nonlocal 'my_list' found
```

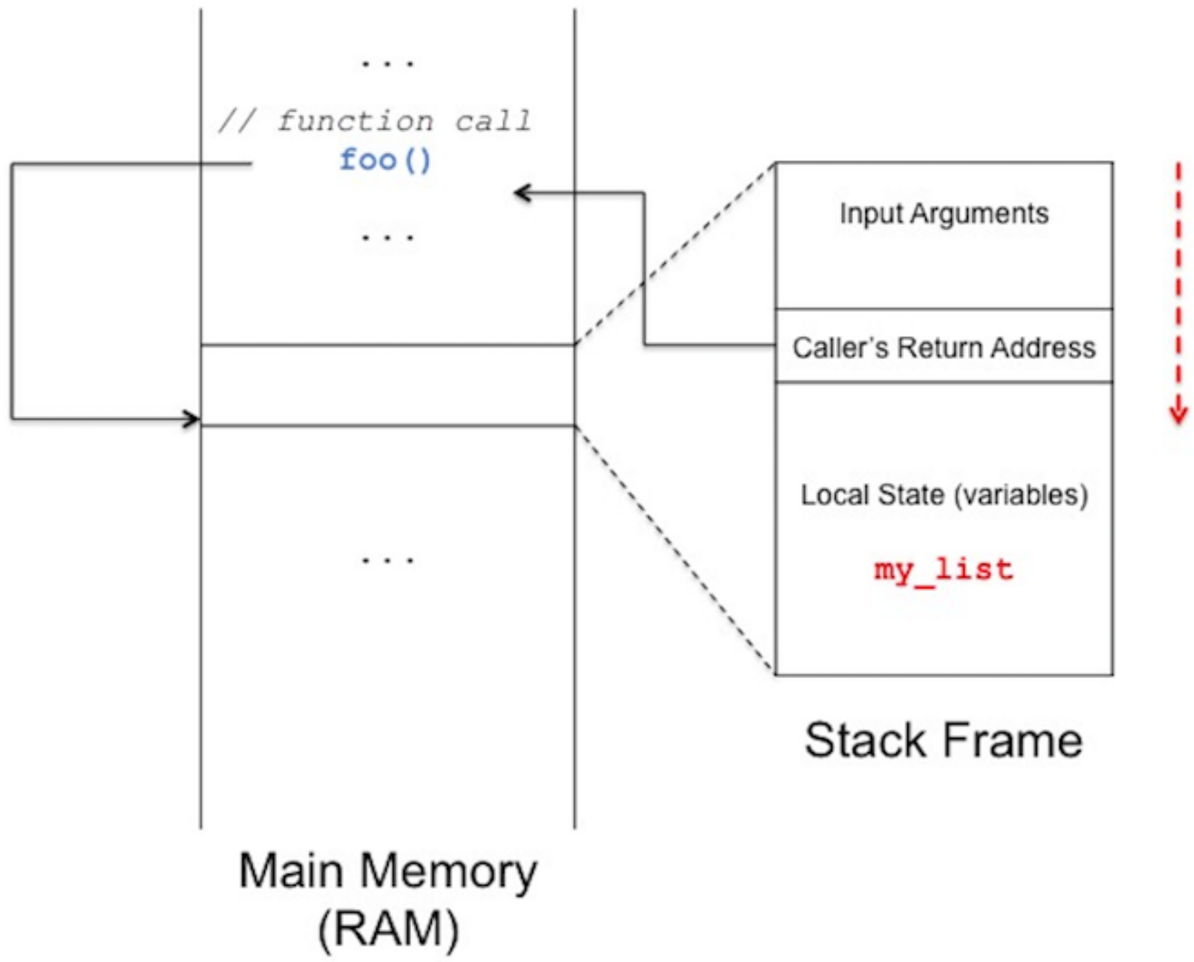
In [9]:
```
# Example 8: correct usage of nonlocal variables
def foo():
    # Define my_list inside the scope of foo() and before calling bar()
    my_list = [36, 49, 64]
    # Print the values contained in my_list before calling bar()
    print("Inside foo() and before calling bar(), my_list is {}".format(my_list))
    def bar():
        # Trying to access my_list defined in foo()'s local scope
        nonlocal my_list
        my_list = [6, 7, 8]
        # Print the values contained in (nonlocal) my_list
        print("Inside bar(), my_list is {}".format(my_list))
    # Calling bar()
    bar()
    # Print the values contained in my_list after calling bar()
    print("Inside foo() and after calling bar(), my_list is {}".format(my_list))

# Define my_list outside the scope of foo() and before calling foo()
my_list = [42, 73, 96]
# Print the values contained in my_list before calling foo()
print("Before calling foo(), my_list is {}".format(my_list))
# Calling foo()
foo()
# Print the values contained in my_list when foo() returns
print("After calling foo(), my_list is {}".format(my_list))
```

```
Before calling foo(), my_list is [42, 73, 96]
Inside foo() and before calling bar(), my_list is [36, 49, 64]
Inside bar(), my_list is [6, 7, 8]
Inside foo() and after calling bar(), my_list is [6, 7, 8]
After calling foo(), my_list is [42, 73, 96]
```

How Function Call (Roughly) Works

- Calling function `foo()` results into the creation of a so-called **stack frame** (a.k.a. **activation frame** or **activation record**).
- The stack frame contains essentially 3 pieces of information:
    - the actual input arguments to the called function (if any);
    - the (memory) address where to return after the called function terminates;
    - the called function's internal state (e.g., local variables).
- Upon returning to the "caller", the stack frame corresponding to the called function is destroyed!

How Function Call (Roughly) Works



NOTE: *When the stack frame corresponding to a function call is destroyed, any variable definition that is in the **local namespace** of the function is destroyed as well **but** if the* `global` *keyword is used to refer to a variable living inside the **global namespace** instead then this will be also visible once the function returns to the caller.*

Returning Multiple Values

- One of the most useful feature of Python functions is their ability to possibly return multiple values
- In many applications (especially, in data science), you will likely encounter many functions that may have multiple outputs.

In [10]:
```
# Consider the following function returning 3 values
def bar():
    x = 3
    y = 4
    z = 5
    return x,y,z

# Assign the values returned by bar() to 3 variables: a, b, and c
a,b,c = bar()
print("a = {0}; b = {1}; c = {2}".format(a,b,c))
```

```
a = 3; b = 4; c = 5
```

In [11]:
```
# Consider the same function returning 3 values
def bar():
    x = 3
    y = 4
    z = 5
    return x,y,z

# We can assign the values returned by bar() to an iterable object (a tuple)
record = bar()
print("a = {0}; b = {1}; c = {2}".format(record[0],record[1],record[2]))
```

```
a = 3; b = 4; c = 5
```

Functions *are* Objects

- This allows the expression of many construct that are difficult to do in other programming languages.

In [12]:
```
# Consider the following example, where we do some data cleaning
# and need to apply a bunch of transformations to a ("messy") list of strings
states = ['   Alabama ', 'Georgia!', 'CAlifoRnia', 'texas!!', 'FlOrIda',
          'south carolina##', 'West virginia?']
# Lots of so-called preprocessing steps need to be done
# to make this list of strings uniform and ready for analysis:
# e.g., whitespace stripping, removing punctuation symbols, and proper capitalization
```

In [13]:
```python
# 1st Solution
# The following Python statement tells the interpreter to import a specific module
# In particular, re stands for the regular expression module (more on import later)
import re

# Define a function that takes as input a list of strings and returns another list
# where strings are properly preprocessed and normalized
def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip() # remove leading and trailing whitespaces
        value = re.sub('[!#?]', '', value) # remove punctuation
        value = value.title() # capitalize only the first letter
        result.append(value)
    return result

# Call the function defined above
clean_strings(states)
```

Out[13]:
```
['Alabama',
 'Georgia',
 'California',
 'Texas',
 'Florida',
 'South Carolina',
 'West Virginia']
```

In [14]:
```
# 2nd Solution
# An alternate approach is to make a list of the operations to apply to the list of strings
# functional pattern (exploiting the fact that functions are objects)

# Define a function for removing punctuation from a string
def remove_punctuation(value):
    return re.sub('[!#?]', '', value)
# Create a list of operations: either built-in functions (e.g., str.title) or user-defined
clean_ops = [str.strip, remove_punctuation, str.title]

# Define our preprocessing function so that it takes the list of functions as 2nd argument
def clean_strings(strings, ops):
    result = []
    for value in strings: # loop through each string as above
        for function in ops: # loop through each operation (function)
            value = function(value) # function is the reference to a function object
        result.append(value)
    return result
```

```
# Call the function defined above (this time with the list of operations)
clean_strings(states, clean_ops)
```

Out[14]:
```
['Alabama',
 'Georgia',
 'California',
 'Texas',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Anonymous (*lambda*) Functions

- Python has support for so-called ***anonymous*** or ***lambda*** functions.
- Those are just functions consisting of a single statement, the result of which is the return value.
- They are defined using the `lambda` keyword.
- They are very convenient in data analysis because in many cases data transformation functions will take functions as arguments.

In [15]:
```
# In the following example, we define a single-statement function in 2 ways:
# 1) The usual way as below
def short_foo(x):
    return x * 2

# 2) Using anonymous function via the lambda keyword
lambda_short_foo = lambda x: x * 2
```

In [16]:
```
# Consider the following function which takes as argument another function foo
# foo is in turn applied to each element of the list
def apply_to_list(some_list, foo):
    return [foo(x) for x in some_list]

# values is the input list
values = [4, 0, 1, 5, 6]
# When calling apply_to_list, instead of defining explicitly foo just use lambda
apply_to_list(values, lambda x: x * 2)
```

Out[16]:
```
[8, 0, 2, 10, 12]
```

Checkpoint Quiz

How would you implement `apply_to_list` function without using `lambda`?

In [ ]:
```
def apply_to_list_no_lambda(some_list):
    '''
    TODO: given the input list some_list, return another list
    so that its i-th element is equal to the i-th element of some_list yet multiplied by 2
    '''
```

Extended Function Call Syntax: `*args`, `**kwargs`

- The way that function arguments work under the hood in Python is actually very simple.
- When you write `foo(a, b, c, d=d_value, e=e_value)`, ***positional*** arguments are packed into a `tuple`, whilst ***keyword*** arguments into a `dict`.
- So, the internal function receives a `tuple args` and `dict kwargs`.

In [17]:
```
# Consider the following function definition
# This takes as input another function foo, plus the list of positional arguments (*args)
# and the list of keyword arguments (**kwargs)
def say_hello_then_call_foo(foo, *args, **kwargs):
    print('args = {}'.format(args))
    print('kargs = {}'.format(kwargs))
    print("Hello! Now I'm going to call '{}({}, {}, {})'".format(foo.__name__, args[0], args[1
    return foo(*args, **kwargs)

# This is the function which will be input to the above
def bar(a, b, c=1):
    return (a + b) / c

# Call say_hello_then_call_foo with bar
say_hello_then_call_foo(bar, 5, 7, c=4.0)
```

```
args = (5, 7)
kargs = {'c': 4.0}
Hello! Now I'm going to call 'bar(5, 7, 4.0)'
```

Out[17]:
```
3.0
```

Iterators

- Having a consistent way to iterate over sequences, like objects in a list is an important Python feature.
- This is accomplished by means of the ***iterator protocol***, a generic way to make objects ***iterable***.

In [18]:
```
# For example, iterating over a dict yields the dict keys
a_dict = {'x': 1, 'y': 2, 'z': 3}
# When you write the loop below, the Python interpreter first attempts
# to create an iterator out of a_dict
for key in a_dict:
    print(key)

# An iterator is any object that will yield objects to the Python interpreter
# when used in a context like a for loop.
# Most methods expecting a list or list-like object will also accept an iterable object
dict_iterator = iter(a_dict)
dict_iterator
```

```
x
y
z
```

Out[18]:
```
<dict_keyiterator at 0x105793ef8>
```

Generators

- A ***generator*** is a concise way to construct a new iterable object.
- Typically, functions execute and return one or more values as a whole.
- Generators, instead, are functions that return a sequence of values *lazily*, pausing after each one until the next one is requested.
- To create a generator function, `yield` keyword rather than `return` is used.

In [19]:
```
# Consider the following example
# Note that squares(n) uses yield instead of return
def squares(n=10):
    print('Generating squares of numbers from 1 to {0}'.format(n))
    for i in range(1, n + 1):
        yield i ** 2

# Check this is actually a generator
squares_gen = squares(12)
print(type(squares_gen))

# Until you request elements from the generator this won't execute its code

# Use the generator
for x in squares_gen:
    print(x, end=' ') # end=' ' Appends a whitespace instead of a newline
```

```
<class 'generator'>
Generating squares of numbers from 1 to 12
1 4 9 16 25 36 49 64 81 100 121 144
```

I/O

I/O Basics

- In this class, we will be using high-level tools provided by third-party modules (e.g., `pandas`) to load data files from disk into Python data structures.
- Still, it is important to understand the basics of how to work with files in Python!

- To open a file for reading or writing, use the built-in `open()` function with either a relative or absolute file path.
- The full API specification for `open()` can be found here.

In [20]:
```
# Assuming we have a file 'sample.txt' stored at this location (relative path)
path = './data/sample.txt'
# The result of the 'open' function is a 'handle' to a file object
# By deafult, file is opened in 'read-only' text mode (binary mode is also available)
# In text mode, if encoding is not specified the encoding used is platform dependent:
# locale.getpreferredencoding(False) is called to get the current locale encoding
infile = open(path) # This is equal to open(path, 'r')
```

Mode	Description
`r`	Read-only mode
`w`	Write-only mode. Creates a new file (erasing the data for any file with the same name)
`x`	Write-only mode. Creates a new file, but fails if the file path already exists.
`a`	Append to existing file (create it if it does not exist)
`r+`	Read and write
`b`	Add to mode for binary files, that is `'rb'` or `'wb'`
`t`	Text mode for files (automatically decoding bytes to unicode). This is the default if not specified. Add `t` to other modes to use this, that is `'rt'` or `'xt'`

In [21]:
```
# infile can be thought of as a reference to a list of (Unicode) strings
# You can loop through each line of the file (delimited by '\n') as follows
for line in infile:
    print(line)
```

```
Sueña el rico en su riqueza,

que más cuidados le ofrece;


sueña el pobre que padece

su miseria y su pobreza;


sueña el que a medrar empieza,

sueña el que afana y pretende,

sueña el que agravia y ofende,


y en el mundo, en conclusión,

todos sueñan lo que son,

aunque ninguno lo entiende.
```

In [22]:
```
# The following is used to remove EOL character from each line
lines = [line.strip() for line in open(path)]
print(lines)
```

```
['Sueña el rico en su riqueza,', 'que más cuidados le ofrece;', '', 'sueña el pobre que pa
dece', 'su miseria y su pobreza;', '', 'sueña el que a medrar empieza,', 'sueña el que afa
na y pretende,', 'sueña el que agravia y ofende,', '', 'y en el mundo, en conclusión,', 't
odos sueñan lo que son,', 'aunque ninguno lo entiende.', '']
```

In [23]:
```
# Whatever we have to do with a file's handle (reading from it or writing to it),
# eventually we should call the 'close()' method on it
infile.close()
```

In [24]:
```python
# Instead of explicitly opening/closing the file's handle, Python has this nice solution
with open(path) as infile:
    for line in infile:
        if line.strip() != '':
            print(line.strip())
```

```
Sueña el rico en su riqueza,
que más cuidados le ofrece;
sueña el pobre que padece
su miseria y su pobreza;
sueña el que a medrar empieza,
sueña el que afana y pretende,
sueña el que agravia y ofende,
y en el mundo, en conclusión,
todos sueñan lo que son,
aunque ninguno lo entiende.
```

Reading from Files

- For readable files, some of the most commonly used methods are `read()`, `seek()`, and `tell()`.
- `read()` returns a certain number of characters from the file.
- What constitutes a "character" depends on whether the file is opened in **text** or **binary** mode.

In [25]:
```
# Example of using 'read' method
# Re-open the previously closed file in text mode (by default, UTF-8 encoding is assumed)
text_file = open(path) # This is equal to open(path, 'r')
# Read 10 characters
print(text_file.read(10))
```

```
Sueña el r
```

In [26]:
```
# What if we use the 'read' method when the file is opened in binary mode
binary_file = open(path, 'rb')
# Read 10 characters
print(binary_file.read(10))
```

```
b'Sue\xc3\xb1a el '
```

In [27]:
```
# The 'read' method advances the file handle's position by the number of bytes read.
# The 'tell' method gives you the current position
# This is the output when this is called on a file opened in text mode
print("Number of bytes read = {}".format(text_file.tell()))
# This is the output when this is called on a file opened in binary mode
print("Number of bytes read = {}".format(binary_file.tell()))
# The difference is due to the fact that the text-mode file handle
# needs to advance by 11 bytes in order to decode 10 characters
# (which is assumed to be UTF-8-encoded by default). The extra byte originates
# from the fact that the'ñ' character is encoded with 2 bytes using UTF-8
# With byte-mode file handle, instead, the number of characters read
# also correspond to the number of bytes read!
```

```
Number of bytes read = 11
Number of bytes read = 10
```

In [28]:
```
# Lastly, 'seek' changes the file position to the indicates byte in the file
# Text-mode file handle
text_file.seek(3)
print("Current byte position (text-mode opened) = {}".format(text_file.read(1)))
# Byte-mode file handle
binary_file.seek(3)
print("Current byte position (binary-mode opened) = {}".format(binary_file.read(1)))
# Finally, close both file handles
text_file.close()
binary_file.close()
```

```
Current byte position (text-mode opened) = ñ
Current byte position (binary-mode opened) = b'\xc3'
```

Writing to Files

- To write text to a file, you can use either the file's `write()` or `writelines()` methods.
- For example, we could create a version of the previous text file with no blank lines.

In [29]:
```
# Write each line of the original file to a new temporary file
with open('data/tmp.txt', 'w') as outfile:
    outfile.writelines(line for line in open(path) if len(line) > 1)

# See if the written file contains what we actually expect
with open('data/tmp.txt') as infile:
    print(infile.readlines())
```

```
['Sueña el rico en su riqueza,\n', 'que más cuidados le ofrece;\n', 'sueña el pobre que pa
dece\n', 'su miseria y su pobreza;\n', 'sueña el que a medrar empieza,\n', 'sueña el que a
fana y pretende,\n', 'sueña el que agravia y ofende,\n', 'y en el mundo, en conclusió
n,\n', 'todos sueñan lo que son,\n', 'aunque ninguno lo entiende.\n']
```

Common I/O Methods

Method	Description
`read([size])`	Return data from file as a string, with optional `size` argument indicating the number of bytes to read
`readlines([size])`	Return list of lines in the file, with optional `size` argument
`readlines([size])`	Return list of lines (as strings) in the file
`write(str)`	Write passed string to file.
`writelines(strings)`	Write passed sequence of strings to the file.
`close()`	Close the handle
`flush()`	Flush the internal I/O buffer to disk
`seek(pos)`	Move to indicated file position (integer).
`tell()`	Return current file position as integer.
`closed`	`True` if the file is closed.