#ThinkPython

File IO

When you run a program, the code instructions and data is stored in RAM. RAM is **volatile** memory. When the program ends or the computer powers off, the contents of RAM is cleared.

When your program saves data to a file, it is written to **non-volatile** storage on a disk.

A **file** is related data stored in a named location on a non-volatile medium such as a disk.

The `open()` function

Python's `open()` function returns a file object that points to a file on a disk. It takes two arguments, the path to a file and the mode in which it should open the file. The default mode is `'r'` for read mode. Other modes are `'w'` for write mode and `'a'` for append mode. Write mode overwrites the content of existing files while append mode adds the new content to the end of existing content in a file.

There is also a `'b'` mode that, when combined with each of the other modes, allows for reading and writing to binary files.

Here is an example:

```
with open('myfile.txt', 'w') as file:
    file.write("Hello, world!')

with open('myfile.txt', 'r') as file:
    print(file.read())
```

In the code above, the `with` keyword creates a block statement that limits the file object's scope and while ensuring the file is properly closed once the scope has been executed.

File Object Methods

The file object has the following methods for accessing the text in a file:

- `.read()` returns the entire contents of the file as a string
- `.readline()` returns every character from the current position in the file up to and including the new line character
- `.readlines()` returns a list where each item is a line from the file
- `.write()` writes a string buffer to the file
- `.close()` closes the file object

File Paths

The `open()` function requires an explicit path to a file. Paths may be absolute (starting from the root directory) or relative (starting from the current working directory).

> absolute path `"/Users/sally/myfile.txt"`

> relative path `"data/myfile.txt"`

File Systems and Directories

Non-volatile storage media is organized by a set of rules called a **file system**. File systems are comprised of files and **directories**. Directories are named containers for files and other directories.

A path to a file includes all the directories and subdirectories the file is contained within. Directories in a path are delimited with slashes. Unix-like (POSIX) operating systems (i.e. Linux and macOS) use forward slashes `/` to delimit paths. Windows uses backslashes `\` to delimit paths.

The `os` module

Python includes the `os` module with many useful functions for working with paths and directories.

The `os.getcwd()` function will return the current working directory as an absolute path.

The `os.path.join(path, file)` function returns a path made from a path and filename that is correctly delimited for the operating system where the code is running.

The `os.listdir(path)` function returns a list of files and directories within the directory passed as an argument.

The `os.path.exists(path)` returns True if the path or file exists. The `os.path.isdir(path)` returns True if the argument is a legitimate path.

The `os.name` statement will return the name of the operating system. Unix-like operating systems will be identified as `posix` and the Windows operating system will be identified as `nt`.

Catching Exceptions

Exceptions are events that stop the execution of a program.

The `try` statement block will attempt the statements in its scope and execute it's `except` clause statements when exceptions occur.

```
try:
    file = open('bad_file.txt')
except:
    print('An error occured.')
```

Databases

A **database** is a file that is organized for storing data.

Python's `dbm` module is used for storing string values in a database object using keys similar to a dictionary data type.

```
import dbm
db = dbm.open('data', 'c')
db['name'] = 'Gordon'
db['occupation'] = 'Engineer'
db.close()
```

The `'c'` argument passed to the `dbm.open()` function means the database file will be created automatically if it does not already exist.

Python's `pickle` module allows almost all data types to be stored in a database as a serialized string.

Use the `pickle.dumps()` function to serialize the data type and use the `pickle.loads()` function to de-serialize the data type back into its original form.

```
import pickle
t1 = (1, 2, 3)
s = pickle.dumps(t1)
t2 = pickle.loads(s)
```

Note: the pickle module is not secure.

Pipes

A **pipe object** allows a string containing a command to be executed by the operating system's CLI shell (command interpreter).

The `os.popen()` function accepts a string and returns a pipe object. The pipe object may be read like a file object.

```
import os
command = 'ls -l' # Unix command to list the directory contents
pipe_obj = os.popen(command)
print(pipe_obj.read())
pipe_obj.close()
```

Modules

A **module** is a file consisting of constants, functions, and classes that may be imported into other programs.

The `import` statement is used to include a module's code in another program.

When a module is imported, test code may be excluded by using the following if statement:

```
if __name__ == "__main__":
    # test code below
```

The value of the special `__name__` variable is only `"__main__"` when the module is executed directly as a program. When the module is imported into another program, the value of the `__name__` variable is the name the file excluding the `'.py'` file extension.

Object-Oriented Programming

Classes and Objects

A **class** is a programmer-defined type. Defining a class creates a **class object**. Creating a new object is called **instantiation**, and the object is an **instance** of the class.

```
class Point:
   ''' Defines a point object. '''
   def __init__(self, x, y):
       self.x = x
       self.y = y

p1 = Point(2, 5) # constructor statement initializes new instance
```

Objects have **attributes** which are variables that are associated with an instance of a class. Values may be assigned to an object's attributes using dot notation (the object's identifier a dot and then the attribute name)

```
p1.x = 3 # p1 identifies the object and x is the attribute
```

An **embedded object** is an object that is assigned as the value of an attribute of another object.

Objects may be the return value of a function. Objects may also be passed by reference as an argument to a function.

Objects are **mutable**. This means an object passed to a function can be modified directly from the scope of that function. This happens because objects are passed by reference and not by value like simple data types such as strings.

When objects are copied the copy is an alias of the original. The `copy` module has functions like `copy.copy(obj)` that creates a **shallow copy** of the object and any references but not its embedded object. There is also a `copy.deepcopy(obj)` function that creates a **deep copy** of an object that also includes embedded objects and the objects they refer to.

Functions

There are two types of functions: pure functions and modifiers

Pure functions do not change the objects they are passed. These functions will often return a new instance of an object.

Modifiers, on the other hand, change the objects they are passed, and as a result often do not return any value.

Class Methods

A **method** is a function that is associated with a particular class.

- methods are defined inside the class definition
- methods are called using dot notation similar to attributes

There are a few special methods commonly used when defining classes.

- The `__init__(self)` method *initializes* an object's attributes. This method is called automatically when an object is first instantiated.
- The `__str__(self)` method returns a string representation of the object. This is used when printing an object or when an object is interpolated in an f-string.
- There are numerous methods such as `__add__(self)` and `__lt__(self)` that are used to **overload** operators such as `+` and `<` operators so that they operate on programmer-defined type operands as well.

Notice that each of these methods have a required first parameter called `self` which refers to the particular object instance that is invoking the method. Other parameters, if needed, should be listed after the `self` parameter.

Decorator are keywords that are prefixed with an @ symbol that modify the behavior of a function. The `@property` decorator allows a method to be used as if it were an attribute.

Polymorphism

Functions that work with several types are called **polymorphic**. Polymorphism promotes code reuse.

Inheritance

Class attributes are variables defined in a class outside of any method and are accessible by any object instance of that class.

A class definition may inherit the attributes and methods of another class.

```
class Hand(Deck):
   # hand class inherits from class Deck
```

Graphical User Interfaces

The `tkinter` module is one of a few modules that may be used to create Graphical User Interfaces with Python.

The import statements commonly used are

```
from tkinter import *
from tkinter import ttk
from tkinter import filedialog
```

To begin, you must create a `Tk()` object to be the main window of your application.

```
window = tk()
window.geometry("400x300')
window.title('Hello, Tkinter!')

mainloop() # keeps the app running
```

Widgets

Graphical objects in `tkinter` are called widgets.

Common widgets:

- `Label()` – displays text content
- `Button()` – used to execute a command
- `Entry()` – single line text field
- `Frame()` – container for other widgets
- `Checkbutton()` – independent option
- `Radiobutton()` – for groups of related options
- `Combobox()` – dropdown list of options
- `Menu()` - window menu bar widget
- `Toplevel()` – creates a new window
- `Text()` – multiline text field
- `Scrollbar()` – vertical or horizontal scroll bar at edge of widget

Widget Options

Widgets may be configured with several options using the `.configure()` method.

```
label = ttk.Label(window, text="Hello, world!")
label.configure(
    foreground='red',
    background='yellow',
    font=('monospace', 14),
    padding=10,
)
```

Geometry Managers

For widgets to appear on the screen they must use a geometry manager. There are three geometry managers: grid, pack, and place. We used grid because it allows us to arrange widgets in rows and columns.

```
label_name = ttk.Label(window, text="Name")
label_name.grid(row=0, column=0)

var_name = StringVar()
entry_name = ttk.Entry(window, textvariable=var_name)
entry_name.grid(row=0, column=1)

button_submit = ttk.Button(window, text="Submit Info")
button_submit.grid(row=1, column=0, columnspan=2)
```

Note in the example above that for each widget, the grid method is called and is passed row and column **keyword** arguments. Also notice the columnspan argument used with the Button widget.

Grid Options

Other options for grid are:

- `padx` – horizontal margin around widget
- `pady` – vertical margin around widget
- `ipadx` – horizontal padding inside widget
- `ipady` – vertical padding inside widget
- `sticky` – assigned `N`, `S`, `E`, `W` to attach the widget to one or more sides of the grid cell

Variable Objects

Variable objects are used in `tkinter` programs to interact with widgets. These variable objects are able to dynamically `.get()` and `.set()` the values associated with a widget.

Variable Object Types:

- `StringVar()`
- `IntVar()`
- `DoubleVar()`
- `BoolVar()`

These are connected to a widget by configuring the widget's textvariable or variable options with the name of the variable object.

Menus

To create a menu you must first configure a menu bar for the top level window object.

```
menu_bar = Menu(window)
window['menu'] = menu_bar
```

Next, you create a drop down menu on the menu bar.

```
menu_file = Menu(menu_bar)
menu_bar.add_cascade(menu=menu_file, label="File")
```

Finally, you add commands to the drop down menu.

```
menu_file.add_command(lable="Quit", command=quit)
```

Note: the command option must be assigned the name of a function (either built-in function or programmer-defined function)

The `Text` widget

The `Text` widget is different than other widgets. It has its own methods for inserting, deleting, and getting the text content it contains.

Loading text from a file:

```
text1 = Text(window, width=60, height=10)
with open('data.txt') as file:
    text1.insert('1.0', file.read()
```

Writing text to a file:

```
with open('data.txt', 'w') as file:
    file.write(text1.get('1.0', 'end')
```

Clearing text:

```
text1.delete('1.0', 'end')
```

The `Filedialog` Module

The `filedialog` module contains several functions that interact with the operating system specific dialogs for opening and saving files.

- `filedialog.askopenfilename()`
- `filedialog.asksaveasfilename()`

Each of these have options like `initialdir` and `filetypes` that modify the default behavior of the dialogs.