Introduction to Programming Lecture 13

- Instructor : Andrew Yarmola andrew.yarmola@uni.lu
- Course Schedule : Wednesday 14h00 - 15h30 Campus Kirchberg B21
- Course Website : sites.google.com/site/andrewyarmola/itp-uni-lux
- Office Hours : Thursday 16h00 - 17h00 Campus Kirchberg G103 and by appointment.

Clone the SciPy Lecture Notes

If you feel like you need more examples, you can always clone the SciPy Lecture Notes

```
git clone https://github.com/scipy-lectures/scipy-lecture-notes.git
```

Each section has some good examples. For instance : `scipy-lecture-notes/advanced/image_processing/examples/` contains a lot of information on advanced image processing.

Final Project Topics

- Numetric PDE Solver and grapher for special types of PDEs.
- A tool for audio syncing using fast Fourier transforms.
- Drawing and zooming Julia (and/or Multibrot) sets.
- A tool for handwritten digit recognition using machine learning or a neural network
- A tool for counting the number and size of objects in an image/video

Work will be in groups and due at the end of January (i.e. January 31st)

Scripts

Scripts are non-graphical standalone programs for doing a specific task. For us, they will be `python` source files, just like modules.

For examples, here are the contents of `three_powers_of_two.py` : a script that prints the first 3 powers of 2.

```
def generate_powers() :
    return [ 2**x for x in range(3) ]

print("Global __name__ is :", __name__)

if __name__ == '__main__' :
    print(*generate_powers(), sep = '\n')
```

Now, if I import this file, nothing will happend except for the fact that I will have `generate_powers` defined.

In [1]:

```
import three_powers_of_two as tpt
```

```
Global __name__ is : three_powers_of_two
```

As you can see, when importing, the global variable `__name__` is set to the filename.

However, if I now go to a console/terminal and **run** the scipt using

```
python three_powers_of_two.py
```

You will see the commands in the `if` statement executed.

```
$ python three_powers_of_two.py
Global __name__ is : __main__
1
2
4
```

So a script can be used both as a module and a tool. However, a program isn't very useful is you can't give it input.

Arguments and Options

The standard way to pass arguments to a script is to give a space separated list after the command call :

```
python three_powers_of_two.py arg1 arg2
```

To read the aguemnts in, we will use the `sys` module's `sys.argv` attribute. We update our `three_powers_of_two_new.py` with :

```
import sys

def generate_powers() :
    return [ 2**x for x in range(3) ]

if __name__ == '__main__' :

    print(sys.argv)

    print(*generate_powers(), sep = '\n')
```

When we run this using the above command in a terminal, we will see :

```
$ python three_powers_of_two.py arg1 arg2
['three_powers_of_two.py', 'arg1', 'arg2']
1
2
4
```

In particular, `sys.argv` is a list that starts with the **name** of the program and then gives **all space separated arugments**.

Remark if you need to have a space in an arugment, you can use (double) quatition marks :

```
$ python three_powers_of_two.py arg1 "arg2 with a space"
['three_powers_of_two.py', 'arg1', 'arg2 with a space']
1
2
4
```

Be very careful with argument spacing when using powerful commands.

Let us make a slightly more useful script `count_vowels.py` that counts vowels in a file

```
import sys

def vowels_in_string(data) :
    return { v : data.count(v) for v in 'aeiou' }

if __name__ == '__main__' :
    if len(sys.argv) < 2 :
        print("Usage: python count_vowels.py file")
        sys.exit(2)

    file_name = sys.argv[1]

    with open(file_name, 'r') as fp :
        data = fp.read()
        print(vowels_in_string(data))
```

Notice that I am doing input checking. This allows me to both inform the user how to use the program and also to check for bad input.

Remark : Make sure your scripts are also useful as **modules** by separating out tasks in your code.

We can run out program to get :

```
$ python count_vowels.py "Lecture 11.ipynb"
{'u': 5197, 'a': 5898, 'i': 5305, 'o': 4934, 'e': 6094}
```

Options and `getopt`

Using `sys.argv` gives us only **positional** arguments for our program. There is a better way using the `getopt` modules. The idea is to specify a **flag** or **keyword** using a `-` or `--` prefix. We would like to do something like this :

```
python hanoi_gif.py -v --d 4 --fps 4 awesome_hanoi_4.gif
```

Let's see a simple example fo how `getopt` works

In [2]:

```
import getopt

argv_list = '-v -d 4 --fps 5 -w something --write nothing arg1 arg2 arg3'.spli
t()

opts, args = getopt.getopt(argv_list, 'vd:w:',
                           ['verbose', 'disk=', 'fps=', 'write='])

print(opts)
print(args)
```

```
[('-v', ''), ('-d', '4'), ('--fps', '5'), ('-w', 'something'), ('-
-write', 'nothing')]
['arg1', 'arg2', 'arg3']
```

As you can see there are **three** types of options/arguments here. The key thing to understand is the line

```
    opts, args = getopt.getopt(argv_list, 'vd:w:',
                               ['verbose', 'disk=', 'fps=', 'write='])
```

The string `'vd:w:'` indicates how to parse **single letter** options preceded by a `-` symbol. While the list `['verbose', 'disk=', 'fps=', 'write=']` indicates how to parse **keyword** options preceded by a `--` symbol

The options here are :

- **flags** : these are the `-v` or `--verbose` options
  - they have no value, their **presence** is all you need
  - they are declared by a letter **without** a colon or a word **without** an =
- **keyword arguments** : these are the `-d`,`-w`, `--disks`, `--write`, and `--fps` options
  - they require a value to follow them when calling the program
  - their declaration is followed by a colon after a letter or an = after a keyword
- **positional arguments** : there are `arg1`, `arg2`, and `arg3`
  - must **follow** all flag and keyword arguments

When `getopt.getopt` parses `argv_list`, it returns a **list of tuples** and a **list of strings**. The list of tuples is map of options to their values and the list of strings is the list of positional arguments.

To apply this to function arguments, we simply need to call `getopt` on `sys.argv[1:]` (dropping the program name).

We can now implement a `hanoi_gif.py` containing the following code. Pay close attention to how I interpret the contents of `opts` and `args`.

```
import sys, hanoi, getopt
import moviepy.editor as mpy

def make_clip(n, fps) :
    hanoi_states = hanoi.solution_states(n)
    hanoi_frames = [ hanoi.frame(n, s) for s in hanoi_states ]
    clip = mpy.ImageSequenceClip(hanoi_frames, fps = fps)
    return clip

def print_usage() :
    print("Usage: python hanoi_gif.pu [-v,--verbose]"
          "[-d,--disks <num_disks>] [--fps <fps>] output_file")

if __name__ == '__main__' :
    try:
        opts, args = getopt.getopt(sys.argv[1:],
                                   'vd:',['verbose','disks=','fps='])
    except getopt.GetoptError as err:
        print(err)
        print_usage()
        sys.exit(2)

    if len(args) != 1:
        print_usage()
        sys.exit(2)
    clip_name = args[0]

    verbose = False
    fps = 2
    n = 3
    for opt, val in opts:
        if opt in ('-v', '--verbose'):
            verbose = True
        if opt in ('-d', '--disks'):
            n = int(val)
        if opt in ('--fps'):
            fps = int(val)

    if verbose : print("Generating clip")
    clip = make_clip(n, fps)
    if verbose : print("Clip created. Writing to file {}.".format(clip_
name))
    clip.write_gif(clip_name)
    if verbose : print("Done.")
```

Remark : It is always better to use program arguments instead of interactive input for a program! However, if there is a time where you need to ask the user something (e.g. should the program delete something), you can use the `input()` method discussed in Lecture 6.

Graphical User Interface (GUI)

Scripts are great, but they can be hard to use for people who aren't comfortable with the command line. Therefore, you might want to write a graphical interface for your program. This is not easy and takes a lot of code. However, much of this code is reusable and repetitive.

There are many different programming languages and tools for GUI applications and `python` is probably not the best one. However, you can make simple interactive programs that work everywhere fairly easily using the package called `tkinter`.

Remark : `tkinter` is the standard tool included with `python` for GUI applications. However, there are many many other libraries for GUI programming that are more modern looking. See https://docs.python.org/3/library/othergui.html#other-gui-packages as well as Kivy and PyjamasDesktop.

To get started we will create a file `basic_tk.py` with the following code :

```
from tkinter import *
# ttk is a slightly more modern look
# calling like this overrides older
# tkinter code
from tkinter.ttk import *

import getopt

class Basic_App_Window(Frame) :

    def __init__(self, master = None):
        super().__init__(master)
        self.initialize()
        self.setup_interface()

    def initialize(self) :
        try:
            opts, args = getopt.getopt(sys.argv[1:],'a:',['action_text=
'])
        except getopt.GetoptError as err:
            print("Usage : python basic_tk.py [-a, --action_text] <text
>.")
            self.master.quit()

        self.action_text = "Hello Everyone!"
        self.display_text = StringVar()
        for opt, val in opts:
            if opt in ('-a', '--action_text'):
                self.action_text = val

    def setup_interface(self) :
        # self.master is set in super().__init__(master)
        self.master.minsize(width = 400, height = 200)
        self.pack(fill = BOTH, expand = True)

        button = Button(self, text = "Push Me")
        button['command'] = self.toggle_text
        button.pack(anchor = NW)

        quit_button = Button(self, text = 'Quit',
                                         command = self.master.quit)
        quit_button.pack(side = 'bottom')

        action_label = Label(self, textvariable = self.display_text)
        action_label.pack(side = 'top')

    def toggle_text(self) :
        if self.display_text.get() == '' :
            self.display_text.set(self.action_text)
        else :
            self.display_text.set('')

if __name__ ==  '__main__' :
    root = Tk()
    app = Basic_App_Window(root)
    app.mainloop()
```

Try runing this from a terminal/console with

```
python basic_tk.py -a "So much code for so little functionality"
```

Much of this code should be self-explanatory, but there are some large concepts.

- The object `root` is the main interface manager. There can only be one in the application.
- The `class Basic_App_Window` is a special type of a `Frame`, which is an area that displays content and can have other subframes.
- When creating objects such as buttons or labels, the first argument is the *master* or *parent* parent container (i.e. `Frame`).
- The `.pack()` command gives instructions on the **Pack** window geometry manager on how to work with the object under resizing. There is also a more complicated **Grid** window geometry manager.

For a decent reference on `tkinter` in `python 3.5` check out http://www.tkdocs.com/tutorial/index.html and https://docs.python.org/3/library/tk.html

Remark : Notice that I am not using properties here because GUI applications tend to have many attributes and it's too wordy to create a property for each.

Remark : If you need to create a **bunch** of similar fields, you can use code such as **loops**!

Canvas

There is a special type of frame called a **canvas** where you can draw geometric objects. Let us add one by updating `setup_interface` by appending the following code

```
def setup_interface(self) :
        # earlier code
        self.canvas.create_oval(100,100,140,140)
        self.canvas.create_line(0,0,100,400)
        self.canvas.create_polygon(40,30,150,50,20,120)
        self.canvas.create_arc(160,50,240,140,
                                start = 0, extent = 150,
                                fill = 'blue')
        self.gif = PhotoImage(file = 'sinc.gif')
        self.canvas.create_image(180, 100, anchor = NW, image = self.gi
f)
```

Plots inside `tkinter`

We can also insert `matplotlib` plots into `tkinter` applications as follows. Let's create a new program called `plot_tk.py` with the code :

```python
from tkinter import *
from tkinter.ttk import *
import numpy as np

import matplotlib
matplotlib.use('TkAgg')

from matplotlib import pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2TkAgg

class Plot_Window(Frame) :

    def __init__(self, master = None):
        super().__init__(master)
        self.initialize()
        self.setup_interface()

    def initialize(self) :
        self.data = np.random.randn(100)

    def setup_interface(self) :
        self.master.minsize(width = 400, height = 200)
        self.pack(fill = BOTH, expand = True)

        self.fig = plt.figure()
        plt.plot(self.data)

        self.plt_canvas = FigureCanvasTkAgg(self.fig, master = self)
        self.plt_canvas.get_tk_widget().pack(fill = BOTH, expand = True)

        self.toolbar = NavigationToolbar2TkAgg(self.plt_canvas, self )
        self.toolbar.pack(side = 'bottom', expand = True)
        self.toolbar.update()

if __name__ ==  '__main__' :
    root = Tk()
    app = Plot_Window(root)
    app.mainloop()
```

Interactive input

There are two general types of input : **pointer input** and **text field input**. The latter is much easier, so let's look at a quick example.

Text entry input

Let's make a basic calulator called `calc.py` with the code :

```
from tkinter import *
from tkinter.ttk import *

class Calc_Window(Frame) :

    def __init__(self, master = None):
        super().__init__(master)
        self.setup_interface()

    def evaluate_and_show(self, event) :
        val = str(eval(event.widget.get()))
        self.result.set('Result : ' + val)

    def setup_interface(self) :
        self.master.minsize(width = 400, height = 200)
        self.pack(fill = BOTH, expand = True)

        instr = Label(self,
                text = "Type your expression and hit enter to evaluate"
,
                wraplength = self.master['width'])

        instr.pack(side = 'top', expand = True)

        field = Entry(self)
        field.bind('<Return>', self.evaluate_and_show)
        field.pack(side = 'top', expand = True)

        self.result = StringVar()
        res_label = Label(self, textvariable = self.result)
        res_label.pack(side = 'top', expand = True)

if __name__ ==  '__main__' :
    root = Tk()
    app = Calc_Window(root)
    app.mainloop()
```

Above, you might notice the `bind` method. This allows one to associate a function callback with a user interface event! For a reference on event types in `tkinter` check out http://effbot.org/tkinterbook/tkinter-events-and-bindings.htm and http://www.tcl.tk/man/tcl8.5/TkCmd/bind.htm#M7.

Pointer input

Pointer input is more complicated. We have to deal with ButtonPress, ButtonRelease, and Motion!. The first two are not too complicated, so let's to a basic example with motion.

Make a file called `paint.py` with the following code

```
from tkinter import *
from tkinter.ttk import *

class Paint_Window(Frame) :

    def __init__(self, master = None):
        super().__init__(master)
        self.setup_interface()

    def paint(self, event) :
       x1, y1 = ( event.x - 1.5 ), ( event.y - 1.5 )
       x2, y2 = ( event.x + 1.5 ), ( event.y + 1.5 )
       event.widget.create_oval( x1, y1, x2, y2, fill = 'green' )

    def setup_interface(self) :
        self.master.minsize(width = 400, height = 200)
        self.pack(fill = BOTH, expand = True)

        instr = Label(self,
                text = "Draw on the canvas using your mouse",
                wraplength = self.master['width'])

        instr.pack(side = 'top', expand = True)

        canvas = Canvas(self)
        canvas.pack(fill = BOTH, expand = True)
        canvas.bind('<B1-Motion>', self.paint)

if __name__ ==  '__main__' :
    root = Tk()
    app = Paint_Window(root)
    app.mainloop()
```

As you play with this code, you might notice that the motion event is very discrete. If you want, you can try to figure out how to draw smooth lines.