```
from tkinter import messagebox as mBox
from tkinter import Tk
root = Tk()
#This keeps the top level window from being drawn
root.withdraw()

mBox.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2016.')
#mBox.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:\nWarning: There might be a bug in this code.')
#mBox.showerror('Python Message Error Box', 'A Python GUI created using tkinter:\nError: Houston ~ we DO have a serious PROBLEM!')
```

Useful References:

Python GUI Programming Cookbook

By Burkhard A. Meier

On safari: http://ncmd-ebooks-1.ncmd.nsa.ic.gov/9781785283758

Great overview of different widgets and also using "themed" `tk` widgets or `ttk` widgets.

Tkinter GUI Application Development Blueprints

By: Bhaskar Chaudhary

On safari: https://ncmd-ebooks-1.ncmd.nsa.ic.gov/9781785889738

Not all of the packages used in the examples are available, but this book is still a great reference for how to build and structure larger GUI projects.

Programming Python, 4th Edition

By: Mark Lutz

On safari: https://ncmd-ebooks-1.ncmd.nsa.ic.gov/9781449398712

Not a dedicated GUI book, this book does have several chapters cover different GUI aspects. It spends more time explaining the underlying logic of GUI's and covering special cases that can trip you up. It starts from the basics, but moves quickly, so I might not be the best resource for total novices.

Python GUI Programming Cookbook

0

(b)(3)-P.L. 86-36

Updated over 1 year ago by [redacted]

3 4 385 98

tk gui python fcs61 tkinter safari

(U) Code from "Python GUI Programming Cookbook"

Recommendations

Python GUI Programming Cookbook

By Burkhard A. Meier

On safari: http://ncmd-ebooks-1.ncmd.nsa.ic.gov/9781785283758

```
#=====================================================================
#Imports
#=====================================================================
import tkinter as tk                    # basic gui
from tkinter import ttk                 # "themed tk" improved gui options
from tkinter import scrolledtext        # For scrolling text boxes
from tkinter import Menu                # For creating menus
from tkinter import Spinbox             # For creating spin boxes
from tkinter import messagebox as mBox  # For message boxes

#=====================================================================
# Initial framework setup for window, tabs, frames, etc.
#=====================================================================

win = tk.Tk()                           # Create instance
win.title("Python GUI")                 # Add a title
win.iconbitmap(r'U:\private\anaconda3\DLLs\pyc.ico')  #Change icon

tabControl = ttk.Notebook(win)          # Create Tab Control

tab1 = ttk.Frame(tabControl)            # Create a tab
tabControl.add(tab1, text='Tab 1')      # Add the tab

tab2 = ttk.Frame(tabControl)            # Add a second tab
tabControl.add(tab2, text='Tab 2')      # Make second tab visible

tab3 = ttk.Frame(tabControl)            # Add a third tab
tabControl.add(tab3, text='Tab 3')      # Make second tab visible

tabControl.pack(expand=1, fill="both")  # Pack to make visible

# We are creating a container frame to hold all other widgets in tab1
monty = ttk.LabelFrame(tab1, text=' Monty Python ')
monty.grid(column=0, row=0, padx=8, pady=4)


# We are creating a container frame to hold all other widgets in tab2
monty2 = ttk.LabelFrame(tab2, text=' The Snake ')
monty2.grid(column=0, row=0, padx=8, pady=4)
```

```
#=====================================================================
#Callback functions
#=====================================================================

def clickMe():                          # Function for when button is clicked
    action.configure(text='Hello ' + name.get()+' number '+numberChosen.get()+'! ')

# Set Radiobutton global variables into a list.
colors = ["Pink", "Magenta", "Purple"]

#We have also changed the callback function to be zero-based, using the list instead of module-level global variables.
# Radiobutton callback function
def radCall():
   radSel=radVar.get()
   if   radSel == 0: monty2.configure(text=colors[0])
   elif radSel == 1: monty2.configure(text=colors[1])
   elif radSel == 2: monty2.configure(text=colors[2])

def _quit():
    win.quit()
    win.destroy()
    #exit()

# Display a Message Box
# Callback function
def _msgBox():
    #mBox.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2016.')
    #mBox.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:\nWarning: There might be a bug in this
    #mBox.showerror('Python Message Error Box', 'A Python GUI created using tkinter:\nError: Houston ~ we DO have a serious PRC
    answer = mBox.askyesno("Python Message Dual Choice Box", "Are you sure you really wish to do this?")
    print(answer)

# Spinbox callback
def _spin():
    value = spin.get()
    print(value)
    scr.insert(tk.INSERT, value + '\n')

#=====================================================================
# Create Menu bar
```

```
#=====================================================================
menuBar = Menu(win)                                          #create menu bar
win.config(menu=menuBar)                                     #add menu bar to gui

fileMenu = Menu(menuBar, tearoff = 0)                        #create menu
fileMenu.add_command(label="New")                            #add option to menu
fileMenu.add_separator()                                     #add separator to menu
fileMenu.add_command(label="Exit", command=_quit)            #add option to menu
menuBar.add_cascade(label="File", menu=fileMenu)             #add menu to menu bar

# Add another Menu to the Menu Bar and an item
helpMenu = Menu(menuBar, tearoff=0)                          #create second menu
helpMenu.add_command(label="About", command=_msgBox)         #add menu item
menuBar.add_cascade(label="Help", menu=helpMenu)             #add menu to menu bar

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Contents Tab1
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#=====================================================================
# Create Labels
#=====================================================================

ttk.Label(monty, text="Enter a name:").grid(column=0, row=0,sticky=tk.W)          # text label position (0,0)
ttk.Label(monty, text="Choose a number:").grid(column=1, row=0, sticky=tk.W)       # Another label (with location)

#=====================================================================
# Text entry box
#=====================================================================

# Adding a Textbox Entry widget
name = tk.StringVar()                                    # tk's version of a string (storage variable)
nameEntered = ttk.Entry(monty, width=12, textvariable=name)    # Entry box
nameEntered.grid(column=0, row=1,sticky=tk.W)            # Entry box location
nameEntered.focus()                                      # when app starts, put curser in box

#=====================================================================
# Combo box
#=====================================================================
number = tk.StringVar()                                  # tk string variable to hold numbewr
numberChosen = ttk.Combobox(monty, width=12, textvariable=number, state = 'readonly') #combo box
```

```
numberChosen['values'] = (1, 2, 4, 42, 100)                    # options for combo box
numberChosen.grid(column=1, row=1,sticky=tk.W)                 # location combo box
numberChosen.current(0)


#======================================================================
# Button
#======================================================================
action = ttk.Button(monty, text="Click Me!", command=clickMe)    # create button with text and function for when clicked
action.grid(column=2, row=1,sticky=tk.W)                         # Position Button in second row, second column (zero-based)

#======================================================================
# Spinbox
#======================================================================

# Adding a Spinbox widget
spin = Spinbox(monty, values=(1, 2, 4, 42, 100), width=5, bd=8, command=_spin)          #spinbox set of values
spin.grid(column=0, row=2)

# Adding a second Spinbox widget
spin2 = Spinbox(monty, from_=0, to=10, width=5, bd=8, relief = tk.RIDGE, command=_spin)  #spinbox range of values

#Alternate relief options:
#spin2 = Spinbox(monty, values=(0, 50, 100), width=5, bd=20, relief = tk.FLAT, command=_spin)
#spin2 = Spinbox(monty, values=(0, 50, 100), width=5, bd=20, relief = tk.GROOVE, command=_spin)

spin2.grid(column=1, row=2)

#======================================================================
# Checkboxes
#======================================================================

# Creating three checkbuttons
chVarDis = tk.IntVar()                                          # tk int variable, for check box state
check1 = tk.Checkbutton(monty, text="Disabled", variable=chVarDis, state='disabled') #disabled checkbox
check1.select()                                                 # add checkmark
check1.grid(column=0, row=4, sticky=tk.W)                       # position checkbox. sticky=tk.W means aligned west

chVarUn = tk.IntVar()                                           # another tk in variable for checkbox state
check2 = tk.Checkbutton(monty, text="UnChecked", variable=chVarUn)                  #un-checked checkbox
check2.deselect()                                               # set checkbox to not checked
```

```
check2.grid(column=1, row=4, sticky=tk.W)                       # position checkbox

chVarEn = tk.IntVar()                                           # checkbox int variable for checkbox state
check3 = tk.Checkbutton(monty, text="Enabled", variable=chVarEn)                    #Checked checkbox
check3.select()                                                 # set checkbox to checked
check3.grid(column=2, row=4, sticky=tk.W)                       # position checkbox

#======================================================================
# Scrollbox
#======================================================================

# Using a scrolled Text control
scrolW  = 30                                                     # scrollbox width
scrolH  =  3                                                     # scrollbox height
scr = scrolledtext.ScrolledText(monty, width=scrolW, height=scrolH, wrap=tk.WORD)      # create scroll box
scr.grid(column=0, row = 5, columnspan=3,sticky='WE')           # position scroll box

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Contents Tab2
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#======================================================================
# Radio Buttons
#======================================================================

# create three Radiobuttons using one variable
radVar = tk.IntVar()

#Next we are selecting a non-existing index value for radVar.
radVar.set(99)

#Now we are creating all three Radiobutton widgets within one loop.
for col in range(3):
    curRad = 'rad' + str(col)
    curRad = tk.Radiobutton(monty2, text=colors[col], variable=radVar, value=col, command=radCall)
    curRad.grid(column=col, row=6, sticky=tk.W)

#======================================================================
# Labels in a labelsframe
#======================================================================
# Create a container to hold labels
```

```
labelsFrame = ttk.LabelFrame(monty2, text=' Labels in a Frame ')
labelsFrame.grid(column=0, row=7)                # position with padding

# Place labels into the container element
ttk.Label(labelsFrame, text="Label1").grid(column=0, row=0)
ttk.Label(labelsFrame, text="Label2").grid(column=0, row=1)
ttk.Label(labelsFrame, text="Label3").grid(column=0, row=2)

#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Contents Tab3
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

tab3 = tk.Frame(tab3, bg='purple')
tab3.pack()


#=====================================================================
#Callback functions
#=====================================================================

def checked():

    if chVarCr.get():

        canvas = tk.Canvas(tab3, width=150, height=80, highlightthickness=0, bg='blue')
        canvas.grid(row=1, column=0)
        canvas = tk.Canvas(tab3, width=150, height=80, highlightthickness=0, bg='blue')
        canvas.grid(row=0, column=1)
    else:
        canvas = tk.Canvas(tab3, width=150, height=80, highlightthickness=0, bg='green')
        canvas.grid(row=1, column=0)
        canvas = tk.Canvas(tab3, width=150, height=80, highlightthickness=0, bg='green')
        canvas.grid(row=0, column=1)

#=====================================================================
# Checkbox and canvases
#=====================================================================


for pinkColor in range(0,2):
    canvas = tk.Canvas(tab3, width=150, height=80, highlightthickness=0, bg='pink')
```

```
    canvas.grid(row=pinkColor, column=pinkColor)
chVarCr = tk.IntVar()                                    # another tk in variable for checkbox state
checkCr = tk.Checkbutton(tab3, text="Color", variable=chVarCr,command=checked)                #un-checked checkbox
checkCr.deselect()                                       # set checkbox to not checked
checkCr.grid(row=0,column=0)                        # position checkbox



#======================================================================
# Display GUI
#======================================================================
win.mainloop()
```

Examples of other message boxes

```
from tkinter import messagebox as mBox
from tkinter import Tk
root = Tk()
root.withdraw()

mBox.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2016.')
#mBox.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:\nWarning: There might be a bug in this coc
#mBox.showerror('Python Message Error Box', 'A Python GUI created using tkinter:\nError: Houston ~ we DO have a serious PROBLEM
```

```
from tkinter import *
root = Tk()
root.withdraw()
messagebox.showerror('Python Message Error Box', 'A Python GUI created using tkinter:\nError: Houston ~ we DO have a serious PR
```

```
from tkinter import messagebox as mBox
help(mBox)
```

(b)(3)-P.L. 86-36

Module: Logging

0

Updated over 2 years ago by [redacted] in COMP 3321

3 4 152 46

fcs6 access extra python

(U) Module: Logging

Recommendations

(U) There comes a time in every developer's career when he or she decides that there must be something better than debugging with `print` statements. Fortunately, it only takes about one minute to set up basic logging in Python, which can help you diagnose problems in your program during development, then suppress all that output when you use the program for real. After that, there's no looking back; you might even start to use a real debugger someday. The **logging** module has many advanced configuration options, but you'll probably see benefits even if you don't ever use any of them.

(U) At its heart, logging is a name for the practice of capturing and storing data and events from a program that are not part of the main output stream. Logging also frequently uses the concept of *severity levels*: some captured data indicates serious problems, other data provides useful information, and some details are useful only when trying to track down bugs in the logic. Capturing messages from any or all of the different severity levels depends on who invoked the program, along with how and why it's being run.

(U) The Basics

(U) The **logging** module is included in the standard library. To begin using it with the absolute minimum effort possible, import it and start writing messages at different levels.

```
import logging

logging.warning('You have been warned')

logging.critical('ABORT ABORT ABORT')

logging.info('This is some helpful information')
```

(U) The **logging** module has several levels, including `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`, which are just integer constants defined in the module. Custom levels can bee added with the `addLevelName` method, but the default levels should usually be sufficient. Each of the module-level functions `warning`, `critical`, and `info` is shorthand for a collection of functionality, which can be accessed through separate methods if fine-grained control is needed.

- Create a log message with severity at the level indicated (the method `log(level, message)` could also be used).
- Send that message to the `root` **logger**
- If the `root` logger is configured to accept messages of that level (or lower), send the message to the default **handler**, which formats the message and prints it to the console.

(U) This explains why the call to `logging.info` did not print to the screen: its severity is not high enough. By default, the root logger is set to only handle messages at the `WARNING` level or higher. The level of the root logger can be configured at the module level, but must be done *before* any messages are sent. Otherwise, the level must be set directly on the logger.

```
[(level, getattr(logging,level)) for level in ['DEBUG','INFO', 'WARNING', 'ERROR','CRITICAL']]
```

```
logging.root.getEffectiveLevel()
```

```
logging.root.getEffectiveLevel() == logging.WARNING # True
```

```
logging.root.setLevel(logging.INFO)
```

```
logging.root.getEffectiveLevel() == logging.INFO # True
```

```
logging.info("Now this should get logged.")
```

```
logging.log(21,"This will also get logged at a numbered custom level")
```

```
logging.basicConfig(level=logging.DEBUG)
```

```
logging.root.getEffectiveLevel() == logging.DEBUG # False
```

```
exit()
```

```
# New Session
```

```
import logging
```

```
logging.basicConfig(level=logging.INFO)
```

```
logging.info("This is some information.")
```

```
logging.root.getEffectiveLevel() == logging.INFO # True
```

(U) From all this, a strategy emerges around using **logging** to improve your debugging.

- Instead of `print` statements, add calls to `logging.debug` to your code.
- At the top of your script, use `logging.basicConfig(level=logging.DEBUG)` during development; switch to `level=logging.INFO` or `level=logging.WARNING` for production.
- Optionally, make a command-line option for your script that enables debugging output (e.g. `my_script.py --verbose`).

(U) So far, we have dealt with only the defaults: the logger `logging.root` , along with its associated `Handler` and `Formatter` . A program can create multiple loggers, handlers, and formatters and connect them all together in different ways. Aside from the `StreamHandler` already encountered, the `FileHandler` is also very common; it writes messages to a file named in its constructor. For ease of exposition, we use only StreamHandlers in our examples. Each handler has a `setLevel` method; a handler acts on a message only only if both the `logger` and the `handler` have levels set at or below the severity level of the message. Unless `logging.basicConfig()` has been called, handlers and formatters must be explicitly defined.

```
logging.basicConfig()

warnlog = logging.getLogger('warnings')

warnlog.setLevel(logging.WARN)

infolog = logging.getLogger('info')

infolog.setLevel(logging.INFO)

infolog.info('An informational message')

info_handler = logging.StreamHandler()

infolog.addHandler(info_handler)

qm_formatter = logging.Formatter('%(name)s???%(message)s???')

info_handler.setFormatter(qm_formatter)

info_handler.setLevel(logging.ERROR)

warnlog.warning('There it goes')

warnlog.info('Not there anymore')

infolog.info("It's coming back")

infolog.error("Oops, it didn't make it")
```

(U) In this example, `infolog` has two handlers: the default handler created by `basicConfig` and the explicitly set `info_handler` with its distinctive `???` -inspired formatter. This second handler only logs messages with severity equal to or higher than `logging.ERROR`, even though the `infolog` passes it messages with severity level as low as `logging.INFO`, as can be seen by the fact that the default handler prints these messages.

(U) Advanced Usage

(U) A variety of handlers have been written for different purposes. The `RotatingFileHandler` from the `logging.handlers` submodule automatically starts new log files when they reach a size, and keeps only a specified number of backups. The `TimedRotatingFileHandler` is similar, but time-based instead of size-based. Other handlers write messages to sockets, email, and over HTTP, TCP, or UDP.

(U) Formatters use old-style string formatting with `%`, and have access to a dictionary which contains several interesting properties, including the name of the logger, the level of severity, the current time, and other information about the environment surrounding the logging message. Custom keys can be passed to the formatter with a dictionary passed via the `extra` keyword in `logging.log` or related shortcut methods, e.g. `logging.warning`.

(U) If the configured levels are too restrictive, custom levels can be added. They must be assigned a number, which determines when messages at that level will be handled. No shortcut method is added for custom levels, so calls must be made to the `log` method. Of course, it's easy to add such a shortcut to the Logger class.

```
INFOWARN = 25

logging.addLevelName(INFOWARN, 'INFOWARN')

def infowarn(self, message):
    self.log(INFOWARN, message)

logging.Logger.infowarn = infowarn

logger = logging.getLogger('info.warn')

logger.infowarn("Halfway between info and warning.")
```

Module: Math and More

0

Updated about 2 years ago by [redacted] in COMP 3321

3 171 44

fcs6 python

(U) Module: Math and More

Recommendations

Math and More

The Math Module is extremely powerful... for doing math-y things. This page is mostly to demonstrate some of the neat things Python can do that are math related. We'll start with the math module.

You can take this a step further and investigate `cmath` for complex operations. Most of what is in `math` is also in `cmath`.

math.py

```
import math
print(dir(math))
```

Constants of note

ππ (Greek letter pi): Ratio of circle's circumference to its diameter.

```
math.pi
```

Euler's number, e, (pronounced "Oil-er"): The mathematical constant that is the base of the natural logarithm

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \lim_{n \to \infty} (1 + \frac{1}{n})^n$$

e=∑n=0∞1n!=limn→∞(1+1n)n

```
# Euler's number (pronounced "Oil-er")
# The mathematical constant
math.e
```

Logs and Exponents

```
math.log10(10)
```

```
math.log10(100)
```

```
math.log10(1000)
```

```
math.log10(1)
```

```
math.log10(0.1)
```

```
math.log10(0)
```

```
math.log10(-1)
```

```
2**3
```

```
# built-in
pow(2,3)
```

```
math.pow(2,3) # Converts arguments to floats
```

The following demonstrating modular exponentiation. Some ways are WAY faster than others...

```
pow(2,3,5)
```

```
(2**3) %5
```

```
pow(100,10000,7)
```

```
(100**10000) % 7
```

```
pow(100, 1000000, 7)
```

```
( 100 ** 1000000 ) % 7
```

Trig Functions

Notice argument to the trig function is measured in radians, not degrees.

```
help(math.sin)
```

```
math.sin(0)
```

If radians the following would be 1:

```
math.sin(90)
```

So, we expect the following to be one:

```
math.sin(math.pi/2)
```

And the following should be 0 (halfway around the unit circle):

```
math.sin(math.pi)
```

Is this zero?!!? See the `e-16` at the end? That's pretty close, say within rounding error, of 0. Just be careful that you don't check that it is exactly zero.

```
help(math.isclose)
```

```
print(math.sin(math.pi) == 0)
```

```
print(math.isclose(math.sin(math.pi), 0, abs_tol = 10**-15))
```

```
help(math.degrees)
```

```
help(math.radians)
```

```
math.degrees(math.pi)
```

```
math.radians(45) == math.pi / 4
```

Fun Example:

Save the first 1,000 of the Fibonacci sequence to a list (starting with 1,1 not 0,1).
Iterate over that list and print out how many digits each number is.

```
def fib_list(n = 1000, init = [1,1]):
    '''Returns a list of the first n Fibonacci numbers.'''
    fib_list = init
    for x in range(n-2):
        fib_list.append(fib_list[-1] + fib_list[-2])
    return fib_list
```

```
def fib_lengths(fib_list):
    '''Returns a list containing the number of digits in each fibonacci number.
    It can be calculated this way because the list passed in are integers.
    Use as intended!'''
    return [len(str(int(x))) for x in fib_list]
```

```
a = fib_list()
print(a[:20])
```

```
a_lengths = fib_lengths(a)
print(a_lengths[:20])
```

```
b = fib_list(init = [1.0,1.0])
print(b[:20])
```

```
b_lengths = fib_lengths(b)
print(b_lengths[:20])
```

```
def fib_lengths(fib_list):
  '''Returns a list containing the number of digits in each fibonacci number.'''
  return [1 + math.floor(math.log10(x)) for x in fib_list]
```

```
b_lengths = fib_lengths(b)
print(b_lengths[:20])
```

NumPy

Pronounced "Num - Py"... not like stumpy...

If running through labbench, run the next two lines. If running on jupyter-notebook through Anaconda, you can just import numpy.

```
import ipydeps
```

```
ipydeps.pip('numpy')
```

Now we can import numpy:

```
import numpy as np
```

numpy's main object is called `ndarray`. It is:

- homogeneous
- multidimensional
- array

Meaning, it is a table of elements, usually numbers. All elements are the same type. Elements are accessed by a tuple of positive integers.

Dimensions are called axes.
The number of axes is the rank.

```
t = np.array([1,2,1])
```

```
t
```

```
np.ndim(t) # used to be np.rank, but this is deprecated
```

```
a = np.array(np.arange(15).reshape(3,5))
```

```
a
```

```
np.ndim(a)
```

```
a.shape
```

```
a.ndim
```

```
a.dtype.name
```

```
# size in bytes ... like sizeof operator, but easier to get to.
a.itemsize
```

```
a.size
```

```
type(a)
```

Doc ID: 6689695

We have three ways to access "row" 1 of a:

```
a[1] a[1,] a[1,:]
```

```
a[1]
```

To get column in position 2:

```
a[:,2]
```

To get single element "row" 0, column 2:

```
a[0,2]
```

Creating and modifying array:

```
c = np.array([2,3,4])
```

```
c.dtype.name
```

```
d = np.array([1.2,3.5,5.1])
```

```
d.dtype.name
```

Let's change one element of c:

```
c[1] = 4.5
```

```
c.dtype.name
```

Ut oh, the type didn't change. But we tried to add a float! Let's see what happened:

```
c
```

The array was updated, but the value we were adding was converted to an `int64`. Be careful. The type matters!!

This doesn't work:

```
f = np.array(1,2,3,4)
```

Needs to be this:

```
f = np.array([1,2,3,4])
```

Interprets a sequence of sequences as a two dimensional array. Sequence of sequence of sequences as a three dimenional array... you get the pattern?

```
g = np.array([[1.5,2,3],[4,5,6]])
g
```

Type can be specified at creation time:

```
h = np.array([[1,2],[3,4]], dtype = complex)
print(h)
print()
print(h.dtype.name)
```

Remember c? Here is how we can change it from integers to floats:

```
print(c)
c = np.array(c, dtype = float)
print(c)
c[1] = 4.5
print(c)
```

Suppose you know what size you want, but you want it to have all zeros or ones and you don't want to write them all out. Notice there is one parameter we are passing for the dimensions, but it is a tuple.

```
np.zeros((3,4))
```

```
np.zeros((3,4), dtype = int)
```

```
np.ones((2,3,4), dtype = np.int16)
```

The following is FAST, but dangerous. It does not initialize the entries in the array, but takes whatever is in memory. USE WITH CAUTION.

```
np.empty((2,3))
```

You know range? Well, there is arange. Which allows us to create an array containing a range of numbers evenly spaced.

```
np.arange(10,30,5) # Start, stop, step
```

```
np.arange(0.2,0.3, 0.01) # Can do floats!!
```

Say we don't want to do the math to see what our step should be but we know how many numbers we want... that's what linspace is for! We get evenly spaced samples calculated over the interval. It takes a start, stop and the number of samples you want. There are other optional arguments, but you can figure those out!

```
np.linspace(0,2,9) # 9 numbers evenly spaced numbers between 0 and 2, INCLUSIVE
```

Playing wth "matrices" ... or are they? The following are **not** necessarily the results of matrix operations. What exactly is going on here?

```
a
```

```
a + 1
```

```
a / 2
```

```
a // 2
```

```
pow(2,a)
```

```
b = a // 2
```

```
a + b
```

```
a * b # Not matrix multiplcation
```

```
b.T  # Transpose
```

Matplotlib

```
## Only if you are on labbench. If using anaconda you don't need to do this.
## If you haven't already improt ipydeps, uncomment the next line also before running
# import ipydeps
ipydeps.pip('matplotlib')
```

```
import matplotlib.pyplot as plt
## If you haven't imported numpy, do so!
#import numpy as np
```

Line Plot

```
a = np.linspace(0,10,100)
b = np.exp(-a)
plt.plot(a,b)
plt.show()
```

Histogram

```
from numpy.random import normal,rand
x = normal(size = 200)
plt.hist(x,bins = 30)
plt.show()
```

3D Plot

```
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
ax = fig.gca(projection='3d')
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm)
plt.show()
```

```
plt.plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
plt.plot([1,2,3], [1,4,9], 'rs',  label='line 2')
plt.axis([-2, 5, -2, 10])
plt.plot(a, b, color='green', linestyle='dashed', marker='o', markerfacecolor='blue', markersize=12)
plt.show()
```

Doc ID: 6689695

Sage

To be filled in...

COMP3321: Math, Visualization, and More!

(b)(3)-P.L. 86-36

0

Updated almost 3 years ago by [redacted] in COMP 3321

3 5 473 111

fcs6 classes matplotlib numpy python

(U) Python Math, Visualization, and More!

Recommendations

Python Math, Visualization, and More!

This notebook will give a very basic overview to some mathematical and scientific packages in Python, as well as some tools to visualize data.

Credit:

Much of this material is developed by Continuum Analytics, based on a tutorial created by R.R. Johansson.

What is NumPy?

Numpy is a Python library that provides multi-dimensional arrays, matrices, and fast operations on these data structures.

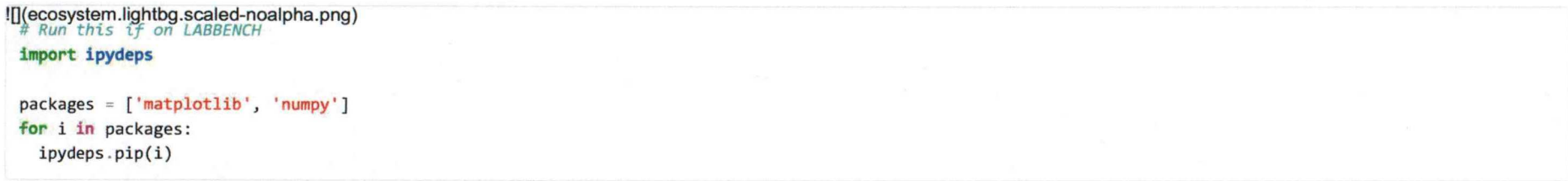
NumPy arrays have:

- fixed size
  - all elements have the same type
    - that type may be compound and/or user-defined
  - fast operations from:
    - vectorization — implicit looping
    - pre-compiled C code using high-quality libraries
      - NumPy default
      - BLAS/ATLAS
      - Intel's MKL

NumPy's Uses and Capabilities

- Image and signal processing
- Linear algebra
- Data transformation and query
- Time series analysis
- Statistical analysis

Numpy Ecosystem




```
# Run this if on LABBENCH
import ipydeps

packages = ['matplotlib', 'numpy']
for i in packages:
  ipydeps.pip(i)
```

Let's install necesary packages

```
import numpy as np
import matplotlib as mpl
import numpy.random as npr
vsep = "\n-------------------\n"
```

`matplotlib` — 2D and 3D plotting in Python

```
# This line configures matplotlib to show figures embedded in the notebook,
# instead of opening a new window for each figure. More about that later.
# If you are using an old version of IPython, try using '%pylab inline' instead.
%matplotlib inline
```

Introduction

Matplotlib is an excellent 2D and 3D graphics library for generating scientific figures. Some of the many advantages of this library include:

- Easy to get started
- Support for LaTeX formatted labels and texts
- Great control of every element in a figure, including figure size and DPI.
- High-quality output in many formats, including PNG, PDF, SVG, EPS, and PGF.
- GUI for interactively exploring figures *and* support for headless generation of figure files (useful for batch jobs).

Matplotlib is well suited for generating figures for scientific publications because all aspects of the figure can be controlled *programmatically*. This is important for reproducibility, and convenient when one needs to regenerate the figure with updated data or change its appearance.

More information at the Matplotlib web page: http://matplotlib.org/

To get started using Matplotlib in a Python program, import the matplotlib.pyplot module under the name plt:

```
import matplotlib.pyplot as plt
```

The matplotlib MATLAB-like API

A great way to get started with plotting using matplotlib is to use the MATLAB-like API provided by matplotlib.
It is designed to be compatible with MATLAB's plotting functions, so if you are familiar with MATLAB, start here.

```
x = np.linspace(0, 5, 100)
y = x ** 2
```

```
x, y
```

```
plt.figure()
plt.plot(x, y, 'g')
plt.xlabel('x')
plt.ylabel('y')
plt.title('title')
plt.show()
```

```
plt.subplot(1,2,1)
plt.plot(x, y, 'r--')
plt.subplot(1,2,2)
plt.plot(y, x, 'g*-');
```

The matplotlib object-oriented API

The main idea with object-oriented programming is to have objects to which one can apply functions and actions, and no object or program states should be global (such as the MATLAB-like API). The real advantage of this approach becomes apparent when more than one figure is created, or when a figure contains more than one subplot.

To use the object-oriented API, we start out very much like in the previous example, but instead of creating a new global figure instance, we store a reference to the newly created figure instance in the `fig` variable, and from it we create a new axis instance `axes` using the `add_axes` method in the `Figure` class instance `fig`:

```
fig = plt.figure()

graph = fig.add_axes([0, 0, 1, 0.3]) # left, bottom, width, height (range 0 to 1)

graph.plot(x, y, 'r')

graph.set_xlabel('x')
graph.set_ylabel('y')
graph.set_title('title');
```

lthough a bit more code is involved, the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure:

```
fig = plt.figure()

graph1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
graph2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# main figure
graph1.plot(x, y, 'r')
graph1.set_xlabel('x')
graph1.set_ylabel('y')
graph1.set_title('Title\n')

# insert
graph2.plot(y, x, 'g')
graph2.set_xlabel('y')
graph2.set_ylabel('x')
graph2.set_title('Inset Title');
```

To save a figure to a file, we can use the savefig method in the Figure class:

```
fig.savefig("filename.png")
```

seaborn — statistical data visualization

Seaborn is a Python visualization library based on `matplotlib`. It provides a high-level interface for drawing attractive statistical graphics.

Homepage for the Seaborn project: http://stanford.edu/~mwaskom/software/seaborn/

bokeh — web-based interactive visualization

Bokeh is a Python interactive visualization library that targets modern web browsers for presentation. Its goal is to provide elegant, concise construction of novel graphics in the style of D3.js, but also deliver this capability with high-performance interactivity over very large or streaming datasets. Bokeh can help anyone who would like to quickly and easily create interactive plots, dashboards, and data applications.

Homepage for Bokeh: http://bokeh.pydata.org/

NumPy Arrays

NumPy arrays (`numpy.ndarray`) are the fundamental data type in NumPy. They have:

- shape
- an element type called *dtype*

For example:

```
M, N = 5, 8
arr = np.zeros(shape=(M,N), dtype=float)
arr
```

Doc ID: 6689695

```
arrzeros = np.zeros(30)
print(arrzeros.dtype, arrzeros.shape)
arrzeros
```

```
arrzeros2 = np.zeros((30,2))
print(arrzeros2.dtype, arrzeros2.shape)
arrzeros2
```

is the NumPy array corresponding to the two-dimensional matrix:


default

NumPy has both a general N-dimensional array *and* a specific 2-dimensional matrix data type. NumPy arrays may have an arbitrary number of dimensions.

NumPy arrays support vectorized mathematical operation.

```
arr = np.arange(15).reshape(3,5)
print("original:")
print(arr)

print()

print("elementwise computed:")
print(((arr+4)*2) % 30)
print(arr.dtype)
```

```
 ((arr.reshape(15,)+4)*2) % 30
```

```
((arr.reshape(5,3)+4)*2) % 30
```

```
arr.reshape(4, 7)
```

Array Shape

Many array creation functions take a shape parameter. For a 1D array, the shape can be an integer.

```
print(np.zeros(shape=5, dtype=np.float16))
```

For nD arrays, the shape needs to be given as a tuple.

```
print(np.zeros(shape=(4,3,2), dtype=float), end=vsep)
print(repr(np.zeros(shape=(4,3,2), dtype=float)))
```

```
print(np.zeros(shape=(2,2,2,2)))
```

Array Types

All arrays have a specific type for their associated elements. Every element in the array shares that type. The NumPy terminology for this type is *dtype*. The basic types are *bool*, *int*, *uint*, *float*, and *complex*. The types may be modified by a number indicating their size in bits. Python's built-in types can be used as a corresponding dtype. Note, the generic NumPy types end with an underscore ("_") to differentiate the name from the Python built-in.

Python Type	NumPy dtype
bool	np.bool_
int	np.int_
float	np.float_
complex	np.complex_

Here is one example of specifying a *dtype*:

```
arr = np.zeros(shape=(5,), dtype=np.float_) # NumPy default sized float
print(arr, "->", arr.dtype)
```

Watch out, though!

```
array = np.array([4192984799048971232, 3, 4], dtype=np.int16)
array
```

```
np.float16('nan')
```

Check the default type...

```
np.array([1], dtype=int).dtype
```

Now we'll take just a moment to define one quick helper function to show us these details in a pretty format.

```
def dump_array(arr):
    print("%s array of %s:" % (arr.shape, arr.dtype))
    print(arr)

vsep = "\n--------------------\n"
```

Array Creation

NumPy provides a number of ways to create an array.

np.zeros and np.ones

```
zrr = np.zeros(shape=(2,3))
dump_array(zrr)
```

```
print(np.ones(shape=(2,5)))
```

```
one_arr = np.ones(shape = (2,2), dtype=int)
dump_array(one_arr)
```

np.empty

`np.empty` is lightning quick because it simply requests some amount of memory from the operating system and then *does nothing with it*. Thus, the array returned by `np.empty` is *uninitialized*. Consider yourself warned. `np.empty` is very useful if you know you are going to fill up all the (used) elements of your array later.

```
# DANGER!  uninitialized array
# (re-run this cell and you will very likely see different values)
err = np.empty(shape=(2,3), dtype=int)
dump_array(err)
```

np.arange

`np.arange` generates sequences of numbers like Python's `range` built-in. Non-integer step values may lead to unexpected results; for these cases, you may prefer `np.linspace` and see below. (For a quick — and mostly practical — discussion of the perils of floating-point approximations, see https://docs.python.org/2/tutorial/floatingpoint.html).

- a single value is a stopping point
- two values are a starting point and a stopping point
- three values are a start, a stop, and a step size

As with `range`, the ending point is *not* included.

```
print("int arg: %s" % np.arange(10), end=vsep)      # cf. range(stop)
print("float arg: %s" % np.arange(10.0), end=vsep) # cf. range(stop)
print("step: %s" % np.arange(0, 12, 2), end=vsep)  # end point excluded
print("neg. step: %s" % np.arange(10, 0, -1.0))
```

np.linspace

`np.linspace(BEGIN, END, NUMPT)` generates exactly *NUMPT* number of points, evenly spaced, on $[\mathrm{BEGIN}, \mathrm{END}]$ [BEGIN,END]. Unlike Python's `range` and `np.arange`, this function is inclusive at BEGIN and END (it produces a closed interval).

```
print("End-points are included:", end=vsep)
print(np.linspace(0, 10, 2), end=vsep)
print(np.linspace(0, 10, 3), end=vsep)
print(np.linspace(0, 10, 4), end=vsep)
print(np.linspace(0, 10, 20), end=vsep)
```

Diagonal arrays: np.eye and np.diag

`np.eye(N)` produces an array with shape (N,N) and ones on the diagonal (an NxN identity matrix).

```
print(np.eye(3))
```

Arrays from Random Distributions

It is common to create arrays whose elements are samples from a random distribution. For the many options, see:

- help(np.random)
- scipy

Uniform on [0,1)

```
print("Uniform on [0,1):")
dump_array(npr.random((2,5)))
```

Standard Normal

`np.random` has some redundancy. It also has some variation in calling conventions.

- `standard_normal` takes *one* tuple argument
- `randn` (which is very common to see in code) takes n arguments where n is the number of dimensions in the result

```
print("std. normal - N(0,1):")
dump_array(npr.standard_normal((2,5)))
print(vsep)
dump_array(npr.randn(2,5)) # one tuple parameter
```

Arrays From a Python List...and a warning!

It is also possible to create NumPy arrays from Python lists and tuples. While this is a nice capability, remember that instantiating a Python list can take relatively long compared to directly using NumPy building blocks. Other containers and iterables will not, generally, give useful results.

```
dump_array(np.array([1, 2, 3]))

print()
dump_array(np.array([10.0, 20.0, 3]))
```

Dimensionality is maintained within nested lists:

```
dump_array(np.array([[1, 2, 3],
                     [4, 5, 6]]))

print()
dump_array(np.array([[1.0, 2],
                     [3, 4],
                     [5, 6]]))
```

Accessing Array Items

Indexing

Items in NumPy arrays may be accessed using a single index composed of multiple values

default

Doc ID: 6689695

```
arr = np.arange(24).reshape(4,6) # random.randint(11, size=(4, 6))

print("the array:")
print(arr, end=vsep)

print("index [3,2] :", arr[3,2], end=vsep)
print("index [3]   :", arr[3], end=vsep)

# non-idiomatic, creates a view of arr[3] then indexes into that copy
print("index [3][2]:", arr[3][2])
```

Compare this with indexing into a nested Python list

```
aList = [list(row) for row in arr]
print(aList)
print(aList[3][2])

try:
    print(aList[3,2])
except TypeError as e:
    print("Unhappy with multi-value index")
    print("Exception message:", e)
```

Slicing

We can also use slicing to select entire row and columns at once:

![default](mef_numpy_slice_01-noalpha.png) ![default](mef_numpy_slice_02-noalpha.png)

Important Differences Between Python Slicing and NumPy Slicing

- Python slicing returns a **copy** of the original data
  - Changing the slice won't change the original.
- NumPy slicing returns a view of the original data
  - Changing the slice **will** change the original data

The NumPy Indexing Page has a lot more information.

```
print("array:")
print(arr)

print("\naccessing a row:")
dump_array(arr[2,:])

print("\naccessing a column:")
dump_array(arr[:,2])

print("\na row:", arr[2,:], "has shape:", arr[2,:].shape)
print("\na col:", arr[:,2], "has shape:", arr[:,2].shape)
```

Bear in mind that numerical indexing will reduce the dimensionality of the array. Slicing from `index` to `index+1` can be used to keep that dimension if you need it.

```
print("lost dimension:", end=' ')
dump_array(arr[2, 1:4])

print("\nkept dimension:", end=' ')
dump_array(arr[2:3, 1:4])
```

Region Selection and Assignment

Multiple slices, as part of an index, can select a region out of an array

```
print("array:")
print(arr)

print("\na sub-array:")
dump_array(arr[1:3, 2:4])
```

Slices are always views of the underlying array. Thus, modifying them modifies the underlying array

```
arr = np.arange(24).reshape(4,6)
print("even elements (at odd indices) of first row:")
print(arr[0, ::2]) # select every other element from first row

arr[0,::2] = -1   # update is done in-place, no copy

print("\nafter assinging to those:")
print(arr)
```

Working with Arrays

Math is quite simple—and this is part of the reason that using NumPy arrays can significantly simplify numerical code. The generic pattern array OP scalar (or scalar OP array), applies OP (with the scalar value) across elements of array.

```
# array OP scalar applies across all elements and creates a new array
arr = np.arange(10)
print("     arr:", arr)
print(" arr + 1:", arr + 1)
print(" arr * 2:", arr * 2)
print("arr ** 2:", arr ** 2)
print("2 ** arr:", 2 ** arr)

# bit-wise ops (cf. np.logical_and, etc.)
print(" arr | 1:", arr | 1)
print(" arr & 1:", arr & 1)

# NOTE:  arr += 1, etc. for in-place
```

```
# array OP array works element-by-element and creates a new array
arr1 = np.arange(5)
arr2 = 2 ** arr1 # makes a new array

print(arr1, "+", arr2, "=", arr1 + arr2, end=vsep)
print(arr1, "*", arr2, "=", arr1 * arr2)
```

Elementwise vs. matrix multiplications

NumPy arrays and matrices are related, but slightly different types.

```
a, b = np.arange(8).reshape(2,4), np.arange(10,18).reshape(2,4)
print("a")
print(a)
print("b")
print(b, end=vsep)
print("Elementwise multiplication: a * b")
print(a * b, end=vsep)
print("Dot product: np.dot(a.T, b)")
print(np.dot(a.T, b), end=vsep)
print("Dot product as an array method: a.T.dot(b)")
print(a.T.dot(b), end=vsep)

amat, bmat = np.matrix(a), np.matrix(b)
print("amat, bmat = np.matrix(a), np.matrix(b)")
print('amat')
print(amat)
print('bmat')
print(bmat, end=vsep)
print("Dot product of matrices: amat.T * bmat")
print(amat.T * bmat, end=vsep)
print("Dot product in Python 3.5+: a.T @ b")
print(amat.T @ bmat)
```

Some Additional NumPy Subpackages

- `np.fft` — Fast Fourier transforms
- `np.polynomial` — Orthogonal polynomials, spline fitting
- `np.linalg` — Linear algebra
- `cholesky, det, eig, eigvals, inv, lstsq, norm, qr, svd`
- `np.math` — C standard library math functions
- `np.random` — Random number generation
- `beta, gamma, geometric, hypergeometric, lognormal, normal, poisson, uniform, weibull`
- many others, if you need it, NumPy probably has it.

FFT

```
PI = np.pi
t = np.linspace(0, 120, 4000)
nrr = np.random.random

signal  =  12 * np.sin(3 * 2*PI*t)   # 3 Hz
signal +=   6 * np.sin(8 * 2*PI*t)   # 8 Hz
signal += 1.5 * nrr(len(t))          # noise
```

```
# General FFT calculation
FFT   = abs(np.fft.fft(signal))
freqs = np.fft.fftfreq(signal.size, t[1] - t[0])
plt.plot(t, signal); plt.xlim(0, 4); plt.show()
plt.plot(freqs, FFT);
```

```
# For one-dimensional real inputs we can discard the negative frequencies
FFT   = abs(np.fft.rfft(signal))
freqs = np.fft.rfftfreq(signal.size, t[1] - t[0])
plt.plot(freqs, FFT); plt.xlim(-0.2, 10);
```

Testing speedup of discarding negative frequencies

```
%%timeit
FFT   = abs(np.fft.fft(signal))
freqs = np.fft.fftfreq(signal.size, t[1] - t[0])
```

```
%%timeit
FFT   = abs(np.fft.rfft(signal))
freqs = np.fft.rfftfreq(signal.size, t[1] - t[0])
```

SciPy - Library of scientific algorithms for Python

The SciPy framework builds on top of the low-level NumPy framework for multidimensional arrays, and provides a large number of higher-level scientific algorithms. Some of the topics that SciPy covers are:

- Special functions (scipy.special)
- Integration (scipy.integrate)
- Optimization (scipy.optimize)
- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fftpack)
- Signal Processing (scipy.signal)
- Linear Algebra (scipy.linalg)
- Sparse Eigenvalue Problems (scipy.sparse)
- Statistics (scipy.stats)
- Multi-dimensional image processing (scipy.ndimage)
- File IO (scipy.io)

Each of these submodules provides a number of functions and classes that can be used to solve problems in their respective topics.

To access the SciPy package in a Python program, we start by importing everything from the `scipy` module...or only import the subpackages we need...

Fourier transform

Fourier transforms are one of the universal tools in computational physics; they appear over and over again in different contexts. SciPy provides functions for accessing the classic FFTPACK library from NetLib, an efficient and well tested FFT library written in FORTRAN. The SciPy API has a few additional convenience functions, but overall the API is closely related to the original FORTRAN library.

To use the `fftpack` module in a python program, include it using:

```
import scipy.fftpack as spfft
```

```
# General FFT calculation
FFT   = abs(spfft.fft(signal))
freqs = spfft.fftfreq(signal.size, t[1] - t[0])
plt.plot(t, signal); plt.xlim(0, 4); plt.show()
plt.plot(freqs, FFT);
```

NumPy FFT vs. SciPy FFT vs. FFTW vs. MKLFFT

Which FFT library should you use?
If you want to use the MKL, you must use NumPy.
The default installations of NumPy and SciPy use FFTPACK
FFTW is faster than FFTPACK, and often faster than MKL

Installing PyFFTW

1. Install FFTW

- apt-get install libfftw3-3 libfftw3-dev
- yum install fftw-devel

1. pip install pyfftw

Interpolation

Interpolation is simple and convenient in SciPy: The `interp1d` function, when given arrays describing X and Y data, returns an object that behaves like a function that can be called for an arbitrary value of x (in the range covered by X). It returns the corresponding interpolated y value:

```
import scipy.interpolate as spinter
```

```
def f(x):
    return np.sin(x)
```

```
n = np.arange(0, 10)
x = np.linspace(0, 9, 100)

y_meas = f(n) + 0.1 * np.random.randn(len(n)) # simulate measurement with noise
y_real = f(x)

linear_interpolation = spinter.interp1d(n, y_meas)
y_interp1 = linear_interpolation(x)

cubic_interpolation = spinter.interp1d(n, y_meas, kind='cubic')
y_interp2 = cubic_interpolation(x)
```

```
fig, ax = plt.subplots(figsize=(10,4))
ax.plot(n, y_meas, 'bs', label='noisy data')
ax.plot(x, y_real, 'k', lw=2, label='true function')
ax.plot(x, y_interp1, 'r', label='linear interp')
ax.plot(x, y_interp2, 'g', label='cubic interp')
ax.legend(loc=3);
```

COMP3321 (U) Python Visualization

0

(b) (3)-P.L. 86-36

Updated over 1 year ago by [redacted] in COMP 3321

3 13 631 246

fcx91 matplotlib bokeh seaborn visualization holoviews

(U) This notebook gives an overview of three visualization methods within Python, matplotlib, seaborn, and bokeh.

Recommendations

Python Visualization

This notebook will give a basic overview of some tools to visualize data. Much of this material is developed by Continuum Analytics, based on a tutorial created by Wesley Emeneker.

First, install necessary packages:

```
# Run this if on LABBENCH
import ipydeps

packages = ['matplotlib', 'seaborn', 'bokeh' ,'pandas', 'numpy', 'scipy',
            'holoviews']
ipydeps.pip(packages)

import numpy as np
import pandas as pd
```

Visualization Choices

There are many different visualization chioces within Python. In this notebook we will look at three main options:

- Matplotlib

- Seaborn
- Bokeh

Each of these options are built with slightly different purposes in mind, so choose the option which best suits your needs!

Matplotlib

Matplotlib is an excellent 2D and 3D graphics library for generating scientific figures. Some of the many advantages of this library include:

- Easy to get started
- Support for LaTeX formatted labels and texts
- Great control of every element in a figure, including figure size and DPI.
- High-quality output in many formats, including PNG, PDF, SVG, EPS, and PGF.
- GUI for interactively exploring figures *and* support for headless generation of figure files (useful for batch jobs).

Matplotlib is well suited for generating figures for scientific publications because all aspects of the figure can be controlled *programmatically*. This is important for reproducibility, and convenient when one needs to regenerate the figure with updated data or change its appearance.

More information at the Matplotlib web page.

To get started using Matplotlib in a Python program, import the matplotlib.pyplot module under the name plt:

```
import matplotlib.pyplot as plt
# This line configures matplotlib to show figures embedded
# in the notebook, instead of opening a new window for each
# figure. More about that later.  If you are using an old
# version of IPython, try using '%pylab inline' instead.
%matplotlib inline
```

The matplotlib MATLAB-like API

A great way to get started with plotting using matplotlib is to use the MATLAB-like API provided by matplotlib. It is designed to be compatible with MATLAB's plotting functions, so if you are familiar with MATLAB, start here.

```
x = np.linspace(0, 5, 100)
y = x ** 2
print(x[0:10])
print(y[0:10])
```

```
plt.figure()
plt.plot(x, y, 'g')
plt.xlabel('x')
plt.ylabel('y')
plt.title('title')
plt.show()
```

```
plt.subplot(1,2,1)
plt.plot(x, y, 'r--')
plt.subplot(1,2,2)
plt.plot(y, x, 'g*-');
```

The `matplotlib` object-oriented API

The main idea with object-oriented programming is to have objects to which one can apply functions and actions, and no object or program states should be global (such as the MATLAB-like API). The real advantage of this approach becomes apparent when more than one figure is created, or when a figure contains more than one subplot.

To use the object-oriented API, we start out very much like in the previous example, but instead of creating a new global figure instance, we store a reference to the newly created figure instance in the `fig` variable, and from it we create a new axis instance `axes` using the `add_axes` method in the `Figure` class instance `fig`:

```
fig = plt.figure()

graph = fig.add_axes([0, 0, 1, 0.3]) # left, bottom, width, height (range 0 to 1)

graph.plot(x, y, 'r')

graph.set_xlabel('x')
graph.set_ylabel('y')
graph.set_title('title');
```

Although a bit more code is involved, the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure:

```
fig = plt.figure()

graph1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
graph2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# main figure
graph1.plot(x, y, 'r')
graph1.set_xlabel('x')
graph1.set_ylabel('y')
graph1.set_title('Title\n')

# insert
graph2.plot(y, x, 'g')
graph2.set_xlabel('y')
graph2.set_ylabel('x')
graph2.set_title('Inset Title');
```

Plotting categorical data

Note: this works in matplotlib 2.0.0; in version 2.1, you can enter the categorical data directly on many of the matplotlib plotting methods.

```
# initialize our data here, turning this into a list of names and a list of counts
data ={'apples': 10, 'oranges': 15, 'lemons': 5, 'limes': 20}
names = list(data.keys())
values = list(data.values())

# first have to create numeric values to cover the axis with the categorical data
N = len(names)
ind = np.arange(N)
width = 0.35

# this will make three separate plots to demonstrate
fig, axs = plt.subplots(1, 3, figsize=(15, 3), sharey=True)
axs[0].bar(ind + width, values)
axs[1].scatter(ind + width, values)
axs[2].plot(ind + width, values)

# here we'll space out the tick marks appropriately and replace the numbers with the names
# for the labels
for ax in axs:
  ax.set_xticks(ind + width)
  ax.set_xticklabels(names)

fig.suptitle("Categorical Plotting")
plt.show(fig)
```

In Matplotlib 2.1+, we can do this directly