Week 13

NumPy and Error Handling

Learning Objectives

ENGINEERING
TEXAS A&M UNIVERSITY

NumPy

- Use NumPy to create an array and perform simple operations on arrays
- Use NumPy with linspace in conjunction with matplotlib to create a plot

Error Handling

- Recognize general types of errors
- Read or write try-except statements for error handling
- Identify and follow code utilizing TypeError, OSError, and ZeroDivisionError exceptions

Engineering Module: NumPy

What is it?

- a library that provides an array object, and an assortment of routines for fast operations on arrays
- Operations are more efficient, and less code, than Python alone
- Many scientific and mathematical packages use NumPy arrays

Creating an array

- You can define each element; the example here creates a 3 x 3 array:

  array_a = numpy.array([[1, 0, 3],[2, 2, 2,], [0, 9 ,3]])

$$\text{array\_a} = \begin{bmatrix} 1 & 0 & 3 \\ 2 & 2 & 2 \\ 0 & 9 & 3 \end{bmatrix}$$

- Or you can have it created as a range of values. This example creates a range of values from 0 to 8, then reshapes into a 3 x 3 array:

  array_b = numpy.arange(9).reshape(3,3)

$$\text{array\_b} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Note: NumPy arrays require the same data type in the array

Math operations on an array

You can perform element-wise arithmetic on an array with `+`, `-`, `*` and `/`

```
c = array_a + array_b
```

You can perform a matrix product using either `.dot()` or the `@` operator

```
d = array_b @ array_a
d = array_b.dot(array_a)
```

You can even use some shortcut operators like `+=` and `*=`, and functions like sum, min, max, average, ... are also available

Indexing and Slicing an array

Essentially identical to indexing and slicing a list in Python for 1-D arrays or a list-of-lists for a multidimensional array.

```
c= array([0, 1, 8, 27, 64, 125])
c[2] = 8
c[2:5] = array([8, 27, 64])
```

Creating an array using linspace

Linspace creates a linearly-spaced array:

numpy.linspace(start, stop, num)

start = starting value of the sequence
stop = the last value of the sequence
num = number of samples to generate (default = 50)

```
my_nums = numpy.linspace(2.0, 3.0, 5)

print(my_nums)

[2., 2.25, 2.5, 2.75, 3.]
```

(an 'array' type)

Creating an array using linspace

```
array_q = numpy.linspace(0, 99, 100)    # unlike 'range', 'linspace' vals are inclusive
for i in array_q:
    print(i)
```

What's going to print?

Creating an array using linspace

```
new_array = numpy.linspace(0, 0.9, 10)
for i in new_array:
    print(i)
```

What's going to print?

Creating an array using linspace

Linspace creates a linearly-spaced array:

```
x_vals = numpy.linspace(0, 0.9, 10)
y_vals = x_vals**2 + 1
print(y_vals)
```

What's going to print?

Creating an array using linspace

Linspace creates a linearly-spaced array:

```
x_vals = numpy.linspace(0, 0.9, 10)
y_vals = x_vals**2 + 1
plt.plot(x_vals, y_vals)
plt.show()
```

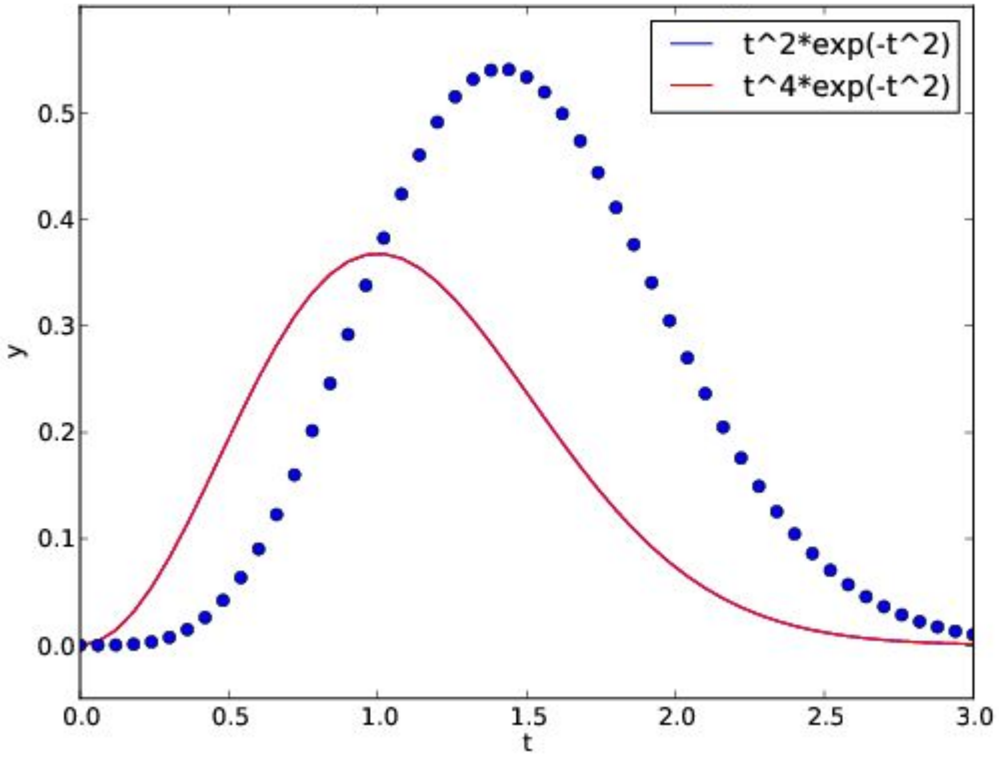
Now what happens?

Plotting using linspace

```
t = numpy.linspace(0, 3, 51)

y1 = t**2*numpy.exp(-t**2)
y2 = t**4*numpy.exp(-t**2)

plt.plot(t, y1, 'r-')
plt.plot(t, y2, 'bo')
plt.xlabel('t')
plt.ylabel('y')
plt.legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
plt.show()
```



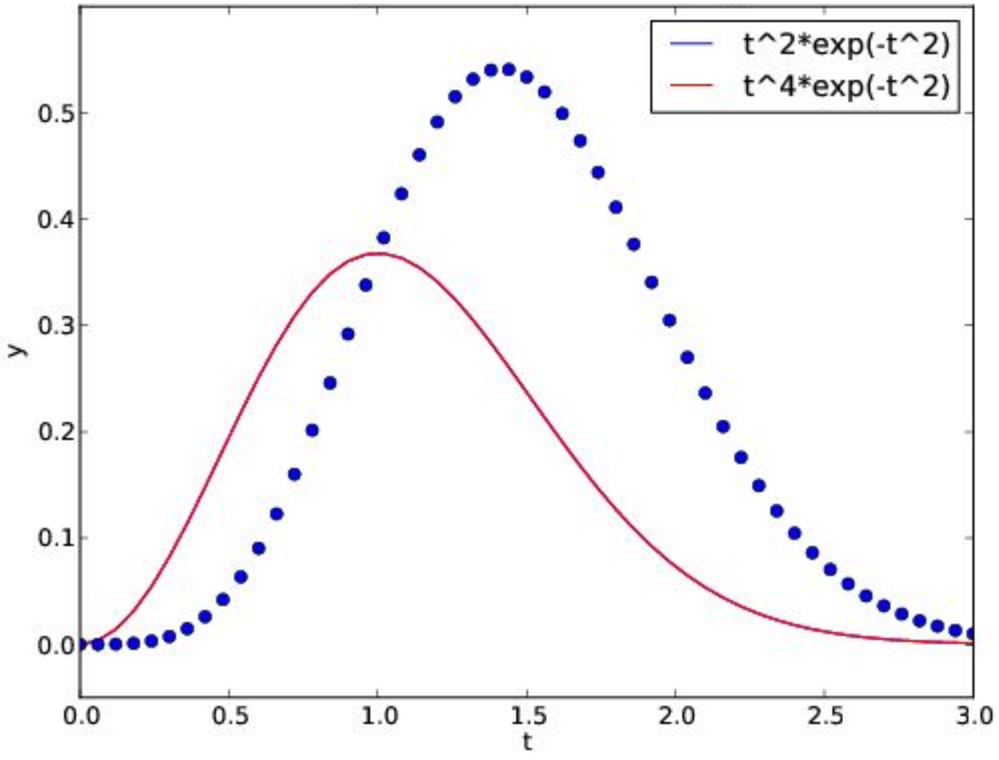
http://hplgit.github.io/primer.html/doc/pub/plot/._plot-readable003.html

Plotting using linspace

```
def f1(t):
    return t**2*numpy.exp(-t**2)
def f2(t):
    return t**2*f1(t)

t = numpy.linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

plt.plot(t, y1, 'r-')
plt.plot(t, y2, 'bo')
plt.xlabel('t')
plt.ylabel('y')
plt.legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
plt.show()
```



http://hplgit.github.io/primer.html/doc/pub/plot/._plot-readable003.html

Types of Error & Handling

Logic, Syntax and Run-Time Errors

Errors in coding

Studies show that better programmers may make fewer errors, but even more significantly can find and fix errors more quickly.

Types of errors: Logic Errors

A mistake in what the code does

- Code may run, but produces the wrong output
- Code may fail on cases that should work

Examples:

```
myanswer = 4          # if 4 should have been assigned to a_new_var instead
if x>2:               # should have been x>=2
    a = 3
b = 4                 # should have been indented
```

Logic errors

Some logic errors are due to typos

- Forgetting to indent
- Misspelling a variable
- Typing the wrong variable name

Some logic errors are due to misunderstandings

- The programmer might have thought that an approach would work, but it doesn't

Tough errors to find

- Error often shows up far from where the mistake actually is

The debugging process will handle these

Types of errors: Syntax Errors

Using incorrect syntax

- Usually the interpreter/IDE will catch this
- Most times, the code will not run at all
- Relatively easy to find and fix most of these

Examples:

```
● whil x>3:                #misspelled while
● if x=2:                  #used assignment (=) instead of equality (==)
● def dosomething(x):
  …
  dosomething()            #forgot parameter
```

Types of errors: Run-Time Errors

- Errors that occur when you run the program (‘at run-time’)
- Often referred to as an “**exception**”
- Might not be predictable ahead of time
- Because Python is interpreted, some syntax errors won’t show up until run-time
- Program uses up too much memory
- Program makes too many nested function calls.
  - Happens with recursion – when a function can call itself.

Run-time error examples

`infile = open("Data.dat",'r')`	`#fails if file doesn't exist`
`answer = int(input('Enter a number:'))`	`#fails if the input is not an integer`
`a = my_list[20]`	`#fails if list has fewer than 21 elements`
`c = a/b`	`#fails if b is 0`

```
L = [1]                                  #fails eventually as list L grows forever
while True:
    L += [1]
```

Handling run-time errors

Some run-time errors are not be predictable (e.g., based on input data), but we can sometimes still find code that is prone to breaking.

Python has a structure for these: the **try-except statement**.

Basic idea:

- Try running code.
- If there is a run-time error, handle the exception (rather than crash)

May offer a chance to fix the problem or exit nicely

- Ask a user for a different file name, or;
- Print out some information about why the program is exiting

The try-except statement

```
try:
    <code to try to run>
except <exception_type>:
    <code to run if there’s an exception
```

We start with the keyword “try”, followed by a colon.

The try-except statement

```
try:
    <code to try to run>
except <exception_type>:
    <code to run if there’s an exception
```

We indent all of the code that we want to try to run.

If there is not an exception (run-time error), then after this code completes, it skips the remainder of the try-except block.

ENGINEERING
TEXAS A&M UNIVERSITY

The try-except statement

```
try:
    <code to try to run>
except <exception_type>:
    <code to run if there’s an exception
```

Next is the except statement.

ENGINEERING
TEXAS A&M UNIVERSITY

The try-except statement

```
try:
    <code to try to run>
except <exception_type>:
    <code to run if there's an exception
```

That is followed by an OPTIONAL exception type. More about this in a second.

The try-except statement

```
try:
    <code to try to run>
except <exception_type>:
    <code to run if there's an exception
```

There is a colon and indentation for the next section of code.

The try-except statement

```
try:
    <code to try to run>
except <exception_type>:
    <code to run if there’s an exception
```

And finally there is code to run if you encounter an exception of the given type

Exception types

These are common exception types (more are available):

- TypeError – an operation or function is applied to an object of inappropriate type
- ValueError – a function receives an argument that has the right type, but an inappropriate value
- OSError – error in dealing with operating system (such as file)
- ZeroDivisionError – error when trying to divide by zero

Example – divide by 0 error

```
a = int(input("Enter a numerator: "))
b = int(input("Enter a denominator: "))
try:
    c=a/b
except ZeroDivisionError:
    print("You can't divide by 0!")
    b = int(input("Enter a different denominator: "))
    c = a/b  # We could have an error here, again!
print(c)
```

Example – error type not specified

```
try:
    L = [1]
    while True:
        L += [1]
except:       # Note: no exception type given. ALL exceptions will come here
    print("Some type of error, but I don’t remember what.")
```

Example

```
age_list = []
def users_age():
    while True:
        try:
            age = int(input("What's your age (in years): "))
            age_list.append(age)
            break
        except ValueError:
            print("Please enter an integer.")
            continue

users_age()
```

Why *ValueError* and not *TypeError*?

Example

```
list = []
def users_name():
    while True:
        try:
            name = str(input("Please enter your first name: "))
            if len(name) > 2 and name.isalpha():
                list.append(name)
                break
            else:
                raise TypeError
        except TypeError:
            if len(name) <= 2:
                print("At least 3 letters, please.")
            else:
                print("Letters only please.")
            continue

users_name()
```