ENGINEERING
TEXAS A&M UNIVERSITY

Week 3

Input, Formatting Output and Calling Modules and Functions

Learning Objectives

Input, Output and Calling Modules and Functions

- Convert between data types in Python, and know when an error will occur
- Find the value of a calculated expression during data type conversions
- Format output in good form using the print command
- Concatenate strings using the + operator
- Modify the print item-separator and line-end commands (`sep = “”`, `end = “”`)
- Use the input command to read information from the user
- Provide a useful prompt to the user with the print command
- Use input information to perform calculations within Python
- Use the newline command (\n) in Python where appropriate
- Define the terms function call and return
- Explain and give examples of the benefits of functions
- Import modules to a program following good programming practice (i.e., don’t use import *)
- Call functions from modules correctly (e.g., math.cos())
- Use a function with or without parameters
- Use a function with or without return variable
- Utilize any function if provided with a function definition or description

Converting between types

- It is possible to convert values of one type into a value of another type.
  - Not for every type combination, though
- General format: `new_type(value)`
  - `value` is a variable, expression, or literal of some type
  - `new_type` is the type to convert into
- Example: converting int to a float:

`float(3)` - this becomes the value 3.0

`x = 2` - x has the integer value 2

`y = float(x)` - y has the float value 2.0, x still has the int value 2

Converting floats to ints

When converting a floating-point number to an integer, the value is truncated (any fractional portion is dropped off)

```
int(2.0)

int(3.14)

int(4.9)

int(0.01)

int(-1.3)

int(-1234.56)
```

Converting floats to ints

When converting a floating-point number to an integer, the value is truncated (any fractional portion is dropped off)

`int(2.0)`	- has the value 2
`int(3.14)`	- has the value 3
`int(4.9)`	- has the value 4
`int(0.01)`	- has the value 0
`int(-1.3)`	- has the value -1
`int(-1234.56)`	- has the value -1234

Converting from strings to ints/floats

Strings, if they “clearly” define an int/float, can be converted to one of those.

```
int('3')

float('3.14')

float('2')

int('2.5')
```

Converting from strings to ints/floats

Strings, if they “clearly” define an int/float, can be converted to one of those.

`int('3')` - this has the integer value 3

`float('3.14')` - this has the floating-point value 3.14

`float('2')` - this has the floating-point value 2.0

`int('2.5')` - this is an error (notice, it does NOT convert to a float, then to an int)

Converting from a number to a string

Does a direct conversion into a string. Floating-point values always have at least one digit before and after the decimal point.

`str(1)` - has the value '1'

`str(2.5)` - has the value '2.5'

`str(1/2)` - has the value '0.5'

`str(10*1.0)` - has the value '10.0'

Note: There is also a “`repr`” alternative to “`str`”. For most types, repr and str work the same, but “repr” lets you convert a string to a string that is printed as shown (with quotation marks, new lines, etc.)

Boolean conversions

Remember, Booleans have the value **True** or **False**

- When converting FROM a Boolean value
  - True is assumed to have the value 1
  - False is assumed to have the value 0
- When converting TO a Boolean value
  - The numeric value 0 has the value False
  - Anything else has the value True

Boolean conversions

```
int(True)

float(True)

float(False)

bool(0)

bool(3)

bool('0')

bool('0.0')

bool('False')
```

Boolean conversions

`int(True)`	- has the value 1
`float(True)`	- has the value 1.0
`float(False)`	- has the value 0.0
`bool(0)`	- has the value False
`bool(3)`	- has the value True
`bool('0')`	- has the value True (is not numeric 0)
`bool('0.0')`	- has the value True (is not numeric 0)
`bool('False')`	- has the value True (is not numeric 0)

Type conversion example

What do you think the value of this expression is?

```
str(float(str(3/2)+str(int(3/2))))*int(int(str(2)+str(7))/int(10.3))
```

Type conversion example

What do you think the value of this expression is?

```
str(float(str(3/2)+str(int(3/2))))*int(int(str(2)+str(7))/int(10.3))

str(float(str(1.5)+str(int(1.5))))*int(int('2'+'7')/int(10.3))

str(float(str(1.5)+str(1)))*int(int('27')/10)

str(float('1.5'+'1'))*int(27/10)

str(float('1.51'))*int(2.7)

str(1.51)*2

'1.51'*2

'1.511.51'
```

Basics of Output with print()

- As we saw earlier, the basic command for output is the print command.
- The print command formats the output in a readable way
  - By default, it also ends the line it prints on, so the next thing printed will be on the next line
- More than one value can be specified in the parentheses, separated by commas
  - Each thing is printed, separated by a space

`print(2.0,'is',2)` outputs: `2.0 is 2`

Printing strings

- Often, to get the format we want, it’s easiest to create a string ourselves, and print the string.
  - e.g. if we don’t want spaces separating elements.
- There are a lot of options for formatting strings in different ways
  - Often helps to line up data or get exact formatting
  - We’ll see some of these in labs throughout the course

Printing strings

If we have values stored in variables x and y, and we want to print: “`<xvalue>:<yvalue>`” (the values, separated by a colon)

```
x=3
y=4
print(x,’:’,y)
print(str(x) + ’:’ + str(y))
```

```
OUTPUT


3 : 4
3:4
```

Notice lack of spaces in second option

Formatting Strings

We can “pad” a string with extra spaces to the left and/or right, using the `.ljust()`, `.rjust()`, and `.center()` commands immediately after a string

```
"2.3".ljust(10)      - Output:   '2.3       '
"2.3".rjust(10)      - Output:   '       2.3'
"2.3".center(10)     - Output:   '   2.3    '
```

Modifying the print() command

By default, the print command

- Separates all the items (listed separated by commas) with a space
- Ends the line after printing (so next thing appears on next line)

We can change how the print command handles those things!

- Item separation, write: `print(…, sep="<something>")`
- Line ending, write: `print(…, end="<something>")`

Note: <something> could be an empty string, if you want nothing printed between items or at the end of a print statement.

Example: Changing print()

What would this output?

```
print("Test",3,5)
print("Test",3,5,sep=',')
print("Test",3,5,sep=',',end=':')
print(15)
```

Example: Changing print()

What would this output?

```
print("Test",3,5)
print("Test",3,5,sep=',')
print("Test",3,5,sep=',',end=':')
print(15)
```

Next

Console

```
Test 3 5
```

The “normal” print output

Example: Changing print()

What would this output?

Next

```
print("Test",3,5)
print("Test",3,5,sep=',')
print("Test",3,5,sep=',',end=':')
print(15)
```

Console

```
Test 3 5
Test,3,5
```

The space separators were replaced by commas

Example: Changing print()

What would this output?

```
print("Test",3,5)
print("Test",3,5,sep=',')
print("Test",3,5,sep=',',end=':')
print(15)
```

Next

Console

```
Test 3 5
Test,3,5
Test,3,5:
```

The space separators were replaced by commas, the end-of-line by a colon

Example: Changing print()

What would this output?

```
print("Test",3,5)
print("Test",3,5,sep=',')
print("Test",3,5,sep=',',end=':')
print(15)
```

Console

```
Test 3 5
Test,3,5
Test,3,5:15
```

There was no new line, so the next statement prints immediately after the colon

Formatting numbers

- Often, we want to output numbers with varying degrees of precision
- This turns out to be a bit more complicated – we will come back to this after seeing some more material
- Look through zyBook 3.11 (Optional for now, but useful)

Getting input

- We’ve seen examples of output – but what about input?
- Input can come from a person typing on the keyboard
  - This input source is referred to in different ways: “standard input” or input from the “console” are two of the more common ones.
  - This is what our current discussion will focus on
- Input can also come from a file, device, another program, etc.
  - We’ll discuss some of this later

The `input()` command

To get input from a user, utilize the `input()` command. We have to assign a variable for the input as well.

example: `user_input_var = input()`

Important: **All input comes in as a string.**

- More correctly, all input from the `input()` command comes in as a string.
- A number input has to be converted from a string to a number.

example: `age = int(input())`

This will read in what is typed, convert it to an integer, and save it in the variable “`age`”.

Input is read until the end of the line is entered

Example: Using input()

Example: a program to compute area of a circle

```
from math import *
r = float(input())
print(pi*r**2)
```

When run, the user will type a number, and the program will output the area of a circle with that radius.

- Try it out!

Note: we have to have the "`from math import *`" line to have `pi` defined.

Example: Using input()

Example: a program to compute area of a circle

```
from math import *
r = float(input())
print(pi*r**2)
```

Console

```
1
3.141592653589793
```

The user typed this in

Example: Using input()

Example: a program to compute area of a circle

```
from math import *
r = float(input())
print(pi*r**2)
```

Not very descriptive. How could you modify the program so that it printed a description of the answer?

Example: Using input()

Example: a program to compute area of a circle

```
from math import *
r = float(input())
print(“The area of the circle is “+str(pi*r**2))
```

Console

```
1
The area of the circle is 3.141592653589793
```

The user typed this in

Example: Using input()

Example: a program to compute area of a circle

```
from math import *
r = float(input())
print(pi*r**2)
```

Better. What change should we make if we want the user to know what they’re about to input?

Example: Using input()

Example: a program to compute area of a circle

```
from math import *
print(“Enter the radius of a circle:”)
r = float(input())
print(“The area of the circle is “+str(pi*r**2))
```

Console

```
Enter the radius of a circle:
1
The area of the circle is 3.141592653589793
```

The user typed this in

Example: Using input()

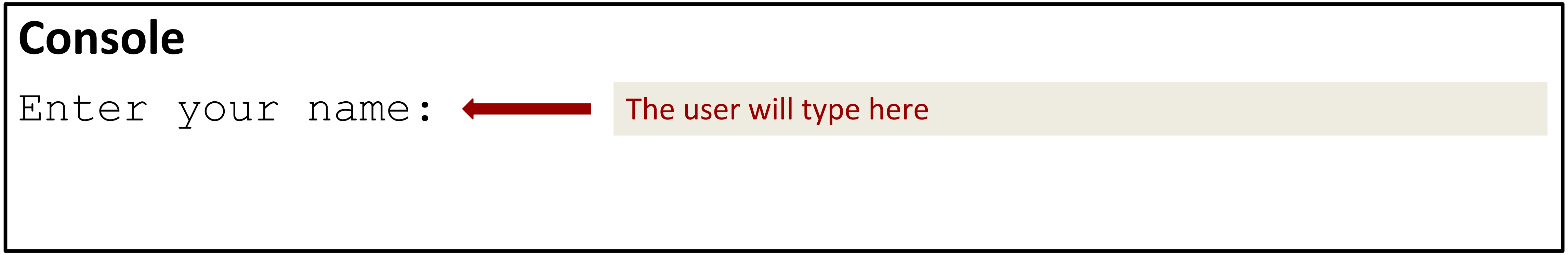
The `input()` command can also print a text prompt itself.

```
user_name = input(x)
```

x can be a literal, variable, or expression

- there is NOT a new line printed after that prompt is printed

Example: `user_name = input("Enter your name: ")`



Console

```
Enter your name:
```

The user will type here

Example: Using input()

Example: a program to compute area of a circle

```
from math import *
r = float(input(“Enter the radius of a circle: “))
print(“The area of the circle is “ + str(pi*r**2))
```

Console

```
Enter the radius of a circle: 1
The area of the circle is 3.141592653589793
```

The user typed the 1

Example: Using input()

Let’s get tricky:

Example: ask user whether they have completed a training assignment (T/F), store answer as a boolean.

Example: Using input()

Let’s get tricky:

Example: ask user whether they have completed a training assignment (T/F), store answer as a boolean.

```
completed_training = input(“Have you completed it? (T / F)”)
```

Does this work? If so, is it a good solution?

Example: Using input()

Let’s get tricky:

Example: ask user whether they have completed a training assignment (T/F), store answer as a boolean.

```
completed_training = bool(input(“Have you completed it? (T / F)”))
```

Does this work? If so, is it a good solution?

Example: Using input()

Let’s get tricky:

Example: ask user whether they have completed a training assignment (T/F), store answer as a boolean.

```
completed_training = bool(input(“Have you completed it? (write
anything for T / leave entirely empty for F)”))
```

Does this work? If so, is it a good solution?

Function calls

Function calls have the form:

```
<function name>(<arguments>)
```

Function calls

Function calls have the form:

```
<function name>(<arguments>)
```

First we have the name of the function

Function calls

Function calls have the form:

```
<function name>(<arguments>)
```

Then, there are parentheses. All function calls have parentheses after the function name.

Function calls

Function calls have the form:

```
    <function name>(<arguments>)
```

Inside the parentheses is, possibly, a list of arguments. Some function calls do not have any arguments.

Arguments are sometimes called “parameters”.

Function calls

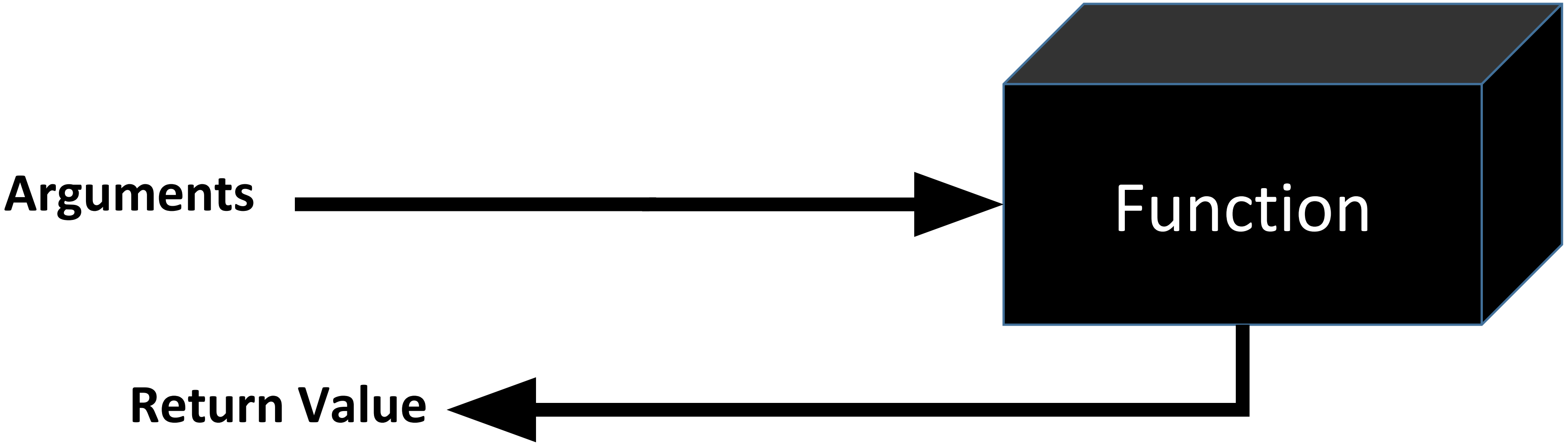
We’ve been seeing many function calls already!

- `print()`
- `input()`
- `int(), float(), str()`
- `open()`
- `len()`
- `sqrt()`

Functions as a “black box”

It may help to think of functions as a “black box”

- They take input (via arguments/parameters)
- They do something (but you don’t need to understand exactly how)
- They return some value



Passing in Data

Some functions need more than one piece of data

- This is usually specified by giving several arguments, separated by commas
- e.g. print(x,y)

Some functions have optional arguments, or even no parameters

- e.g. input(), input("Enter something:")

Returning Data

Many functions return a single value

- Example: sqrt(x) returns the square root of x

Sometimes, we want to return more than one value.

- For this, we need to understand **tuples**
- We'll discuss this in a few weeks

Where do we get functions?

Built-in functions

- Default functions included in Python (e.g., `print()`, `input()`)

Functions we import from modules

- We’ll talk about this in a few slides!

Functions a user writes

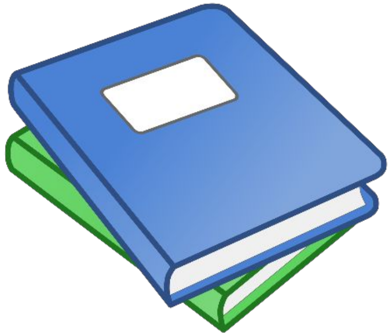
- We’ll talk about this later in the semester

An Analogy

Imagine that you can read and write...

Now ... what if the only books you have are what you wrote yourself?

- Over time you could make a lot of documents containing information you once knew
- But, think of how limited the range of knowledge you could store is!



An Analogy

Now picture a library of information that other people had put together as well

- You have far more information available
- You have less to write yourself

This is the motivation behind **libraries** in programming

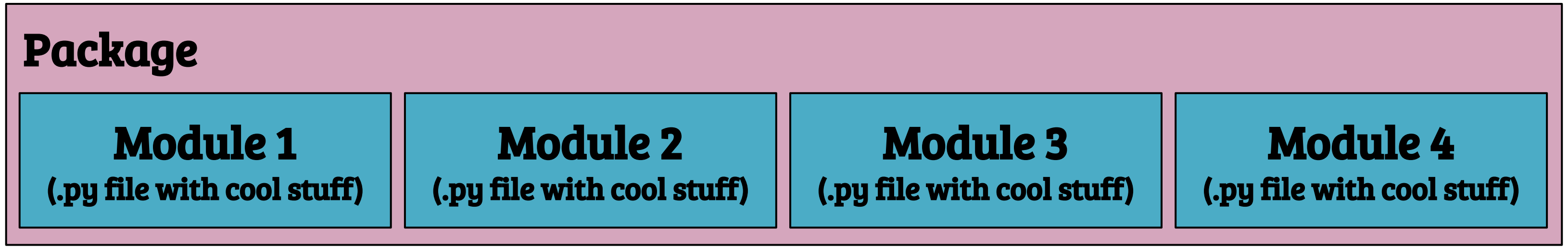
The Man
Ancian Greek for Newbies
SHARKS LOVE PETS
500 Fun Experimentaions 500
Chemistry
MOBY the Whale
Egyptian is the greatest language (after Greek)
THE BLACK TEMPLE
William Wallace
Medics & Me
All your Base Are Belong to Us
DICTIONARY FRENCH ENGLISH DICTIONARY
John Snow
Luke, I am your father !
DICTIONARY
COOKING
A STAR IS BORN
Odyssey of Species
UNDER SIEGE
Heal with Plans
Viserys Targaryen AUTOBIOGRAPHY
Anubis - The God who wants to be loved
RED ROADS
Sorcery for Dummies
Configuring its own Network
The Little House
Pink is not only for Girls
LEONARDO DA VINCI SECRETS
The man who sold the world
My little Bamboo
TRAVEL IN MEXICO

Modules and Packages

In Python, we refer to one of these collections as a **module**

- Basically a single file, containing Python code someone else wrote

Related modules can be put together into a **package**



Package
Module 1
(.py file with cool stuff)
Module 2
(.py file with cool stuff)
Module 3
(.py file with cool stuff)
Module 4
(.py file with cool stuff)

Note: The generic term in programming for this concept is “library”.

Importing Modules

Typically, a module primarily defines functions

- Sometimes other things are also defined, like variables

We've seen an instance of this using the math module

- Functions were defined (e.g., sin, sqrt)
- Constants were defined (e.g., pi, e)

To have access in our own code, we **import** the module

Importing Modules: **the whole module**

To import a whole module, use the command:

```
    import <module name>
```

This makes all functions in that module available in your program.

Importing Modules: the whole module

To import a whole module, use the command:

```
import <module name>
```

This makes all functions in that module available in your program.

To call (use) a function:

```
<module name>.<function name>(<arguments>)
```

Importing Modules: **the whole module**

To import a whole module, use the command:

```
import <module name>
```

This makes all functions in that module available in your program.

To call (use) a function:

```
<module name>.<function name>(<arguments>)
```

The module name is the name of the module.

Importing Modules: **the whole module**

To import a whole module, use the command:

```
import <module name>
```

This makes all functions in that module available in your program.

To call (use) a function:

```
<module name>.<function name>(<arguments>)
```

A period is placed between the module name and the function name. It indicates the function will be from that specific module.

Importing Modules: **the whole module**

To import a whole module, use the command:

```
import <module name>
```

This makes all functions in that module available in your program.

To call (use) a function:

```
<module name>.<function name>(<arguments>)
```

Next is the name of the function itself.

Importing Modules: the whole module

To import a whole module, use the command:

```
   import <module name>
```

This makes all functions in that module available in your program.

To call (use) a function:

```
   <module name>.<function name>(<arguments>)
```

Finally there are the parentheses, possibly containing arguments.

Example

The math module has a function, sqrt, that can be used to compute the square root.

```
import math
a = math.sqrt(2.0)
```

Importing Modules: individual functions

To import individual functions from a module instead, use the command:

```
from <module name> import <function names>
```

Importing Modules: individual functions

To import individual functions from a module instead, use the command:

```
from <module name> import <function names>
```

Start with the command "from"

Importing Modules: individual functions

To import individual functions from a module instead, use the command:

```
from <module name> import <function names>
```

The module name is the name of the module

Importing Modules: individual functions

To import individual functions from a module instead, use the command:

```
from <module name> import <function names>
```

Then the command “import”

Importing Modules: individual functions

To import individual functions from a module instead, use the command:

```
from <module name> import <function names>
```

Finally is the list of functions to import from the module, separated by commas.

Importing Modules: individual functions

To use the functions imported this way, you don't need to list the module name first.

Example:

```
from math import sin, sqrt
a = sqrt(2.0)
b = sin(3.14159/4)
```

Importing Modules: all functions

We can import all the functions from a module by using an * in the place of the list of functions.

- We've been doing this a lot, e.g.: `from math import *`

This is generally frowned upon...

- Functions may have the same name from different modules!
- You probably don't know *everything* you're importing.
- A function overrides an existing function when it loads.
- Instead, use <module>**.**<function> to make it explicit what you're using.

Packages

A package is a collection of individual, related modules. To access a module in a package, use the format:

```
<package name>.<module name>
```

For example, to refer to the module pyplot in the package matplotlib:

```
matplotlib.pyplot
```

Note: The modules in a package are often called “submodules”.

Renaming

You can rename a function by adding “as” on to the end of the from…import command

- To give a more consistent name in your code, or to save typing.

Example, renaming ‘sqrt’ as ‘sr’:

```
from math import sqrt as sr
a = sr(2.0)
```

Getting Modules: Python Library Reference (default)

Many commonly used modules, such as:

- math (math operations),
- cmath (math for complex numbers)
- random (random numbers)
- etc. (more than 200 more)

You still have to *import* them for use.

- See https://docs.python.org/3/library/index.html

Getting Modules: External sources

A list of all registered Python packages and modules is available at the Python Package Index (https://pypi.python.org/pypi)

- These are not necessarily good packages! Users enter what they make into the repository (many are entered daily)
- Some automatically install with certain ‘versions’ of Python.

  e.g., Anaconda Python automatically includes additional modules used with math / science / engineering computations.
- external packages / modules are installed using the pip routine or the conda package manager

Adding modules using pip or Conda

Pip is a routine for installing Python packages. It should be installed with Python, but may need to be updated.

- Run pip from the command line of your system: *cmd* for Windows, *Terminal* for Mac OSX
- Enter `pip install <package name>`
- See documentation: https://pip.pypa.io/en/stable/

Conda is a package manager for use with the Anaconda distribution of Python. It includes graphical and command-line interfaces for installing packages.

- See documentation: https://conda.io/docs/user-guide/

Getting and Using Modules

- As you program in Python, you will probably find there are modules you'd like to use.
- Modules generally have documentation, explaining how they work.
- Most modules have an entire "philosophy" about how the various components they provide should work and be used together.
- When using modules, it is more work to read and understand how a module works than it is to get it installed on your system.

Current Assignments

Complete the CPSC module and Academic Honesty part 1 modules on the eCommunity page

- Due 9/15, by end-of-day

Lab Assignment 3	Due 9/15 by end-of-day
Lab Assignment 3b	Due 9/15 by end-of-day

Preactivity: Complete zyBook chapter 4 and 5 prior to class next week

Pre-Lecture Slides posted on Section Materials page