APS106: Fundamentals of Computer Programming

Arnav Patil

Department of Electrical and Computer Engineering, University of Toronto

Week 1 (Jan 8 - Jan 12)

Chapter 1

1.1 Programming

Basic instruction types are INPUT, PROCESS, and OUTPUT.

Computational thinking - creating a sequence of instructions to solve problems

Algorithms - series of instructions

1.2 Programming using Python

Python interpreter - program executing Python code

Interactive interpreter - allows user to execute one line at a time

Statement - Program instruction

1.3 Development environment

Integrated Development Environment - code development is usually done in an integrated development environment

1.4 Computers and programs

Switches control whether electricity flows through a wire or not.

Processors are circuits created to execute a list of instructions. Memory circuits store lists in addressed locations.

Machine instructions - instructions represented as binary, combine to form executables

Programmers then made assembly language instructions to make code higher level

Compilers - Programs that automatically translate high-level language into executable programs.

1.5 Computer tour

Processor clock ticks measure the number of instructions a processor can execute per second.

In 1958, engineers created the first transistors, which are switches integrated onto a chip.

Moore's Law dictates the number of transistors on an IC has doubled nearly every 18 months since invention.

Cores are multiple processors location on one single chip.

1.6 Language History

Scripting languages execute programs without the need for compilation.

1.7 Why whitespace matters

Whitespace → any block space or newline

2.2 Identifiers

PEP8 (Python Enhancement Proposal) outlines basics of how to write neat and consistent code

2.3 Objects

Represent a value and is automatically created by interpreter

Name binding - process of associating names with an interpreter object

Object mutability tells whether object value is allowed to be changed

2.4 Numeric types: floating point

Floating point numbers have decimals.

Floating point literal is written w/ fraction part, even if it is zero.

Week 2 (Jan 15 - Jan 19)

1.8 Basic input/output

Newline → Output after every print()

Can use end = ‘ ’ inside print() to keep output of next print statement on the same line

Output in same print() command by printing \n

2.8 Heading basics

Module → file containing Python code that can be used by other modules/scripts

Made available for use by import statement

Once imported, any object defined is accessed by using dot notation

3.1 Print functions

A function w/ no return statement is called a void airport

3.2 User-defined function basics

Function definition → consists of function’s name and block of statements

Function call → invocation of the function’s name

Function may return a single value using a return statement

3.3 Dynamic typing

Python uses dynamic typing to determine type of object

Unlike lower-level languages which use static typing

3.6 Functions stub

Incremental development → small amount of code written and tested at a time

Function stub → function definitions whose statements haven’t been written yet

Week 3 (Jan 22 - Jan 26)

4.1 If-else branches (general)

Branches are like steering people to different-sized tables based on group sizes

If → only taken only if an expression evaluates to True

If-else branches can be extended into another structure

4.2 Detecting equal values

Equality operator (==) → evaluates to True if left and right sides are equal

Inequality operator (!=) → evaluates to True if left and right sides are NOT equal

Week 4 (Jan 29 - Feb 2)

7.1 Loops (general)

Program that executes loop’s statements (body)

Each run through the code is called an iteration

7.2 While loops

Repeatedly executes the body the loop while the loop’s evaluation is True.

Week 5 (Feb 5 - Feb 9)

Chapter 2

2.3 Objects

An object represents a value and is automatically created by the interpreter when executing a line of code. Note that programmers do not explicitly create objects, the interpreter creates and manipulates objects as needed to run the given code.

Objects are used to represent everything in a Python program, including integers, strings, functions, lists, etc.

After an object has been used and is no longer necessary, it is automatically deleted from memory and then thrown away. This process of deleting unused objects is called garbage collection.

Name binding is the process of associating names with interpreter objects. An object can have more than one name bound to it, but every name is associated with only one object.

Each Python object has the value:

1. Value: a value such as 20, “abcdef”, etc.
2. Type: the type of the object, such as an integer or string.
3. Identity: a unique identifier that describes the object.

The built-in function `type( )` returns the type of an object.

The object’s type also determines its mutability, which indicates whether the object’s value is allowed to change. Integers and strings are immutable. When either of these are manipulated, a new object with the same name and new given value is generated.

The built-in function `id( )` gives the identity of an object.

Chapter 5

5.1 String basics

A string is a sequence of characters, that can be stored in a variable. A string literal is a string value specified in the source code of the program.

The string type is a special construct known as a sequence type: a type that specifies a collection of objects ordered from left to right.

The `len()` function is used to find the length of a string.

A programmer can access a character at a specific index by appending brackets [ ] containing the index.

Writing or altering individual characters of a string variable is not allowed, since they're immutable. An assignment statement must be used to update an entire string variable.

A program can add new characters to the end of a string in a process known as string concatenation. String concatenation does not contradict string immutability, since new variables are generated.

5.2 String formatting

A formatted string literal, also known as an f-string, allows a programmer to create a string with placeholder expressions that are evaluated as the program executes. An f-string starts with an f character before the starting quote and uses curly braces { } to denote the placeholder expressions. A placeholder expression is also called a replacement field.

```
number = 6
number = 32

print(f'{number} burritoes cost ${amount}')
```

An = sign is provided after the expression in a replacement field to print both the expression and its result, which is a useful debugging technique when dynamically generating lots of strings and output.

A format specification inside a replacement field allows a value's formatting in the string to be customized. This is introduced with a colon in the replacement field, it separates the "what" on the left from the "how" on the right.

A presentation type is part of a format specification that determines how to represent a value in text form, such as an integer, a floating point, and so on. This is also introduced using a colon in the replacement field.

5.9 Type conversions

A type conversion is a conversion of one type to another, such as an integer to a float.

An implicit conversion is a type conversion made automatically by the interpreter.

Chapter 6

6.1 String splicing

Strings are a sequence type, having characters numbered by index from left to right. An index is an integer matching each position in a string's sequence of characters.

Slice notation has the `my_str[start:end]` , which creates a new strong whose value contains the relevant characters in the string.

Omitting a start index yields the characters from indices 0 to the end - 1 spot. Similarly, omitting the end index yields the characters from the start index to the end of the string.

The stride determines how much to increment the index after reading each element.

6.2 Advanced string formatting

A format specification may include a field width, which defines the minimum number of characters that must be inserted into the string.

A format specification can also include an alignment that detrmines how a value should be aligned within the width of the field.

The fill character is used to pad a replacement field when the inserted string is smaller than the field width.

The optional precision component of a format specification indicates how many digits should be included in the output of floating types. The precision follows the field width component in the format specification.

6.3 String methods

The `replace(old,new)` function returns a copy of the string with all occurrences of the substring old replaced by the substring new.

Similarly, the `replace(old,new,count)` does the same but only replaces the first specified number of occurrences.

Some more methods:

`find(x)`	Returns the index of the first occurrence of item x in the string.
`find(x,start)`	Same as above but starts search at the start index.
`find(x,start,end)`	Same as above but ends search at the end index.
`rfind(x)`	Same as above but searches the first occurrence of the string in reverse order.

Methods for creating new strings from an existing string:

`capitalize()`	Returns a copy of the strong w/ the first character capitalized
`lower()`	Returns a copy with all characters in lower case
`upper()`	Returns a copy with all characters in upper case
`strip()`	Removes all leading and trailing whitespaces removed.
`title()`	Returns the string but in title case

6.4 Splitting and joining strings

The `split()` method splits a string into a list of tokens, which are substrings that form the larger string. A separator is a character or sequence that indicated where to split the string into tokens.

The `join()` performs the inverse operation, by joining a list of strings together to create a single string.

6.5 String formatting using %

A string formatting expression allowed a programmer to create a string with placeholders that are replaced by the values of the variables. This placeholder is called a conversion specifier.

6.6 String formatting using `format()`

This method allows a programmer to create a string with placeholders that are replaced by values or variable values at execution.

```
number = 6
amount = 32
```

```
print('{} burritos cost ${}'.format(number,amount))
```

Named replacement allows a programmer to create a keyword argument, which defines a name and value in the format() parentheses. *Good practice is to use named replacement when formatting strings with many replacement fields to make the code more readable.*

Chapter 11

11.1 Readings files

A common programming task is to retrieve input from a file using the `open()` function.

The `file.readline()` method returns a list of strings, where the contents of the file is split into contents of each line as a separate string.

11.4 The ‘with’ statement

A with statement can be used to open a file, execute a block of statements, and automatically close the file when complete. The method creates a context manager, which manages the use of a resource by performing set-up and teardown operations.

Week 6 (Feb 12 - Feb 16)

Chapter 3

3.14 Functions with branches/loops

A function’s block of statements may include branches, loops, and other statements.

Chapter 7

7.5 For loops

A for loop statement loops over each element in a container one at a time, assigning a variable with the next element that can then be used in the loop body.

The container in a for loop is typically a list, tuple, or string.

A for loop may also iterate backward over a sequence, starting at the last element and ending with the first element by using the reversed() function.

7.6 Counting using the `range()` function

Range() generates a sequence of integers between a starting point that is included in the range, and an ending integer that is not included, as well as an integer step value.

7.7 While vs. for loops

A for loop combined with the range() function is generally preferred over while loops.

General rules:

1. Use a for loop when the the number of iterations is computable before entering the loop.
2. Use a for loop when accessing the elements of a container.
3. Use a while loop when the number of iterations is not computable before entering the loop.

7.8 Nested loops

A nested loop is a loop that appears as part of the body of another loop. They are commonly referred to as the inner and outer loops.

7.9 Developing programs incrementally

Experienced programmers practice incremental programming by starting with a simple version of the program and growing the program little by little into a complete version.

A #FIXME comment attracts attention to code that needs to be fixed in the future.

7.10 Break and continue

A break statement in a loop causes to exit immediately. It can also yield a loop that is easier to understand.

A continue statement in a loop causes an immediate jump to the while or for loop header statement. It improves the readability of the loop.

7.11 Loop else

A loop may include an else clause that executes only if the loop terminates normally and doesn't use a break statement.

The loop else construct executes if the loop completes normally.

7.12 Getting both index and value when looping: `enumerate( )`

The `enumerate( )` function retrieves both the index and corresponding element value at the same time, providing a cleaner and more readable solution. It yields a new tuple each iteration of the loop, with the tuple containing the current index and corresponding element value.

Unpacking is a process that performs multiple assignments at once, binding comma-separated names on the left to the elements on the right.

Week 7 (Feb 26 - Mar 1)

Chapter 5

5.3 List basics

A container is a construct used to group related values together and contains references to other objects instead of data.

A list is a container created by surrounding a sequences of variables or literals with brackets [ ].

For example, `my_list = [10, 'abc']` creates a new list. Each item in the list is called an element.

Lists are sequences, meaning the contained elements are ordered by position in the list, known as the element's index, starting with 0.

`my_list = [ ]` creates an empty list.

Individual list elements can be accessed using an indexing expression by using brackets.

Lists, unlike strings and integers, are mutable.

A method instructs an object to perform some action, and is executed by specifying the method name following a '.' symbol and an object. Some commands to add or remove list elements:

`append()`	Add new elements to a list
`pop()`	Used to remove an element from a list
`remove()`	Used to remove an element from a list

Sequence-type functions are built-in functions that operate on sequences like lists and strings. Similarly, there are sequence-type methods built into the class definitions like lists and strings.

Operation	Description
`len(list)`	Finds the length of the list
`list1 + list2`	Produce a new list by concatenating list2 to the end of list1
`min(list)`	Find the element in the list with the smallest value.
`sum(list)`	Find the sum of all elements of a list
`list.index(val)`	Find the index of the first element in the list whose value matches val

Operation	Description
`list.count(val)`	Count the number of occurrences of val in the list.

Chapter 8

8.3 Iterating over a list

Iterating over each element in a list is so common that Python uses a special for loop.

IndexError and enumerate( )

A common error is to try to access the list with an index that is out of the list's range. This causes the program to automatically terminate execution and generate an **IndexError**.

The built in function `enumerate()` iterates over a list and provides an iteration counter.

8.5 List nesting

A list contain a list within itself as an object.

List nesting allows a user to create a multi-dimensional data structure like a matrix.

A programmer can access all of the elements in a nested list by using nested for loops.

8.6 List slicing

A programmer can use slice notation to real multiple elements from a list, creating a new list that contains only the desired elements.

An optional element of slice notation is the stride, which indicates how many elements are skipped between extracted items in the source list.

8.7 Loops modifying lists

Sometimes a program iterates over a list while modifying the elements, such as changing some elements' values or moving elements' positions.

A common error when modifying a list during iteration is to update the loop variable instead of the list.

A common error is to add or remove a list element while iterating over that list.

8.8 List comprehension

The Python language provides a convenient construct, known as list comprehension, that iterates over a list, modifies each element, and returns a new list of the modified elements.

`new_list = [expression for loop_variable_name in iterable]`

A list comprehension has three components:

1. An *expression component* to evaluate for each element in the iterable object.
2. A *loop variable component* to bind to the current iteration element.
3. An *iterable object component* to iterate over (list, string, tuple, enumerate, etc).

8.9 Sorting lists

One of the most useful list methods is `sort()`, which performs an in-place rearranging of the list elements, sorting the elements from lowest to highest.

The `sorted()` built-in function provides the same sorting functionality as the `list.sort()` method, however, `sorted()` creates and returns a new list instead of modifying an existing list.

Sorting also supports the ***reverse*** argument. The reverse argument can be set to a Boolean value, either `True` or `False`. Setting `reverse=True` flips the sorting from lowest-to-highest to highest-to-lowest. Thus, the statement `sorted([15, 20, 25], reverse=True)` produces a list with the elements `[25, 20, 15]`.

Week 8 (Mar 4 - Mar 8)

Chapter 5

5.4 Tuple basics

A tuple stores a collection of data but is **immutable**. Similar to a list, it is also a list type.

Not as common as a list in practical usage but can be useful when a programmer wants to ensure that values do not change.

A named tuple allows the programmer to define a new simple data type that consists of named attributes.

```
from collection import namedtuple

Car = namedtuple('Car', ['make', 'model', 'price', 'horsepower',

chevy_blazer = Car('Chevrolet', 'Blazer', 32000, 275, 8)

print(chevy_blazer)

>>> Car(make='Chevrolet', model='Blazer', price=32000, horsepowe
```

5.5 Set basics

A set is an unordered collection of unique elements. It has the following properties:

1. Elements are unordered: elements in the set do not have a position index or value.
2. Elements are unique: no elements in the set share the same value.

A set that can be created using the `set( )` function, which accepts a sequence-type iterable object. A set literal can be written using curly braces $\{\ \}$ which commas separating set elements.

Sets are mutable - elements can be added or removed using set methods.

5.6 Dictionary basics

A dictionary is a Python container used to describe associative relationships. A dictionary is represented by the dict object type.

A key is a term that can be located in a dictionary. A value describes some data associated with a key, such as a definition.

A dict object is created using curly braces to surround the key:value pairs that comprise the dictionary components.

If no entry with a matching key exists in the dictionary, then a **KeyError** runtime error occurs and the program is terminated.

5.7 Common data types summary

Type	Examples
Numeric	Integer Float
Sequence	String List Tuple Set
Mapping	Dictionary

5.9 Type conversions

A type conversion is a conversion of one data type to another.

An implicit conversion is a type conversion automatically made by the interpreter

```
1 + 2 # returns an int type
1 + 2.0 # returns a float type
1.0 + 2.0 # returns a float type
```

Chapter 8

8.12 Dictionaries

Some approaches to create a dict:

1. The first approach wraps braces { } around key-valuue pairs of literals and/or variables creates a dictionary with keys.
2. The second approach uses dictionary comprehension, which evaluates a loop to create a new dictionary, similar to how list comprehension creates a new list. → **OUT OF SCOPE**
3. Other approaches use the dict function

8.13 Dictionary methods

A dictionary method is a function provided by the dictionary type that operates on a specific dictionary object.

8.14 Iterating over a dictionary

A for loop can be used to iterate over a dictionary object, with the loop variable set to a key of an entry in each iteration. The order in which the jeys are iterated is not necessarily the order in which the elements were inserted in the dictionary.

The Python interpreter creates a **hash** of each key, which is a transformation of the key into a unique value that allows the interpreter to perform fast lookup. So, the ordering is determined is determined by the hash value, but hash values can change depending on the Python version and other factors.

A view object provides read-only access to dictionary keys and values. A program can iterate over a view object to access one key-value pair, one key, or one value at a time, depending on the method used.

8.15 Dictionary nesting

A dictionary may contain one or more nested dictionaries, in which the dictionary contains another dictionary as a value.

A data structure is a method of organizing data in a logical and coherent fashion. Container objects like lists and dicts are already a form of a data structure, but nesting such containers provides a programmer with much more flexibility in the way that the data can be organized.

8.16 String formatting using dictionaries

Mapping keys

Sometimes a string contains many conversion specifiers. Such strings can be hard to read and understand. Furthermore, the programmer must be careful with the ordering of the tuple values in case items are mistakenly swapped.

Week 9 (Mar 11 - Mar14)

Chapter 3

3.15 Multiple function outputs

A return statement that returns multiple outputs, separated by commas, returns them as a tuple by default. It can also be made to return as a list, or another type of variable:

```
return [mean, std_dev] # this return statement outputs a list

return mean, std_dev
return (mean, std_dev) # both of these return tuples
```

Unpacking is an operation that allows a statement to perform multiple assignments at once. These variables come from a tuple or list (ordered containers).

```
average, std_dev = get_grade_stats(student_scores)

# the first, or zeroth, variable in the tuple 'student_scores' g
# average, and the second, or first, variable gets saved to std_
```

3.16 Keyword arguments and default parameter values

Python provides for keyword arguments to allow for users to input their parameters by name rather than an implicit order, which might be confusing if there are a lot of input parameters.

```
def print_car_info(make, model, year, engine)
    # prints car info

print_car_info(make='Ford', model='Escape', year=2018, engine=5
```

Good practice is to use keyword arguments for any function containing more than approximately four arguments.

Sometimes a function has parameters that are optional. A function can have a ***default parameter value*** for one or more parameters, meaning that a function call can optionally omit an argument, and the default parameter value will be substituted for the corresponding omitted argument.

A parameter's ***default value*** is the value used in the absence of an argument in the function call.

3.17 Arbitrary argument lists

A function definition can include an ****args*** parameter that collects optional positional parameters into an ***arbitrary argument list*** tuple.

Adding a final function parameter of *****kwargs***, short for ***keyword arguments***, creates a dictionary containing "extra" arguments not defined in the function definition. The keys of the dictionary are the parameter names specified in the function call.

Week 10 (Mar 18 - Mar 22)

Chapter 9

9.1 Classes: introduction

In programming, an object is a grouping of data (variables) and operations that can be performed on that data (methods).

Abstraction/information hiding

Abstraction occurs when a user interacts with an object at a high level, allowing lower-level internal details to remain hidden (aka ***information hiding*** or ***encapsulation***).

An ***abstract data type*** (***ADT***) is a data type whose creation and update are constrained to specific well-defined operations. A class can be used to implement an ADT.

Python built-in objects

Python automatically creates built-in objects for a programmer to use and include the basic data types like integers and strings.

9.2 Classes: grouping data

The **class** keyword can be used to create a user-defined type of object containing groups of related variables and functions.

The object maintains a set of ***attributes*** that determines the data and behaviour of the class.

```
class Time:
    def __init__(self):
            self.hours = 0
            self.minutes = 0
```

An **instantiation** operation is performed by ‘calling’ the class, using parentheses like a function call as in `my_time = Time( )`.

An instantiation operation creates an instance, which is an individual object of the given class.

A method is a function defined within a class. The **__init__** method, commonly known as a constructor, is responsible for setting up the initial state of the new instance.

Attributes can be accessed using the ***attribute reference operator*** "." (sometimes called the ***member operator*** or ***dot notation***).

9.3 Instance methods

A function defined within a class is known as an instance method. An instance method can be referenced using dot notation.

```
class Time:
    def __init__(self):
        self.hours = 0
        self.minutes = 0

    def print_time(self):
        print(f'Hours: {self.hours}', end=' ')
        print(f'Minutes: {self.minutes}')
```

9.4 Class and instance object types

A ***class object*** acts as a *factory* that creates instance objects. When created by the class object, an ***instance object*** is initialized via the __init__ method.

A ***class attribute*** is shared among all instances of that class. Class attributes are defined within the scope of a class.

9.7 Class interfaces

A class interface consists of the methods that a programmer calls to create, modify, or access a class interface.

Class customization can redefine the functionality of built-in operators like <, >=, +, -, and * when used with class instances, a technique known as ***operator overloading***.

Week 11 (Mar 25 - Mar 29)

Chapter 10

10.1 Modules

A module being required by another program is called a dependency.

Evaluating an import statement initiates the following process to load the module:

1. A check is conducted to determine whether the module has already been imported. If already imported, then the loaded module is used.
2. If not already imported, a new module object is created and inserted in sys.modules.
3. The code in the module is executed in the new module object's namespace.

A dictionary of the loaded modules is stored in ***sys.modules*** (available from the sys standard library module). If the module has not yet been loaded, then a new module object is created. A ***module object*** is simply a namespace that contains definitions from the module. If the module has already been loaded, then the existing module object is used.

10.3 Importing names

A programmer can specify names to import from a module by using the ***from*** keyword in an import statement.

10.6 Packages

A ***package*** is a directory that, when imported, gives access to all of the modules stored in the directory. Large projects are often organized using packages to group related modules.

The *from* technique of importing also works with packages, allowing individual modules or subpackages to be directly imported into the global namespace. A benefit of this method is that higher-level package names need not be specified.

Week 12 (Apr 1 - Apr 5)

Chapter 12

12.1 Introduction to data science

Data science is an interdisciplinary field focused on discovering patterns and describing relationships using data. Data science uses techniques from computer science and statistics.

Data scientists can also build, test, and interpret a ***data model***, a representation of a real-life system that organizes data elements and informs how the elements relate to one another.

Features are recorded for individual ***instances***, or observational units, in the dataset.

Big data describes datasets with large volume, created and updated with high velocity, that have variety in structure and format.

12.2 Data science life cycle

The ***data science life cycle*** is a five-step process for completing a research project using data. In some organizations, data scientists are involved in every step of the data science life cycle.

Step	Description
Step 1: Gathering data	Identify available and relevant data; gather new data if needed.
Step 2: Cleaning data	Reformat datasets, create new features, and address missing values.
Step 3: Exploring data	Create data visualizations and calculate summary statistics to explore potential relationships in the dataset.
Step 4: Modelling data	Use modelling skills and content knowledge to fit and evaluate models, measure relationships, and make predictions.
Step 5: Interpreting data	Describe and interpret conclusions from data through written reports and presentations.

Step 1: Gathering data

Structured data is stored in a pre-defined format, typically with features stored in columns and instances in rows. ***Unstructured data*** does not have a predefined format and is difficult for humans to interpret.

Step 2: Cleaning data

Most software packages and programming languages require datasets to be structured as a table with features in columns and instances in rows. Data scientists combine data from multiple sources or file types while making sure the final dataset is in the proper format.

Step 3: Exploring data

During exploratory data analysis, data scientists use plots and graphs to search for meaning in a dataset. Visualizations help data scientists recognize patterns or trends in a dataset, identify unusual observations, and brainstorm appropriate models. Summary statistics like the mean and median are often calculated during the exploration step.

Step 4: Modelling data

Models that predict a feature with known values in the original dataset are ***supervised models***.

- ***Classification models*** predict categorical features.
- ***Regression models*** predict numerical features.

Unsupervised models look for hidden groups or patterns in the dataset rather than predicting a known feature.

Step 5: Interpreting data

Data scientists must effectively interpret the results of a model and communicate findings to technical and non-technical audiences. After interpreting data, data scientists may decide that a model needs to be adapted for changing circumstances or new data.

12.3 Introduction to Python for data science

Advantages	Disadvantages
Readability: Python reads like English, and functions from the same library use consistent syntax.	Consistency: Different libraries may have different syntax conventions.
Popularity: Python is popular in data science and elsewhere in industry, which means resources for learning Python are widely available.	Memory: Python uses more computer memory than other programming languages.
Innovation: New data science models and technologies are constantly added to Python.	Speed: Other programming languages such as Julia perform computations on datasets more quickly than Python.

Import name	Common alias	Description
numpy	np	NumPy includes functions and classes that aid in numerical computation. NumPy is used in many other data science packages.
pandas	pd	pandas provides methods and classes for tabular and time-series data.
sklearn	sk	scikit-learn provides implementations of many machine learning algorithms with a uniform syntax for preprocessing data, specifying models, fitting models with cross-validation, and assessing models.
matplotlib.pyplot	plt	Matplotlib allows the creation of data visualizations in Python. The functions mostly expect NumPy arrays.
seaborn	sns	seaborn also allows the creation of data visualizations but works better with pandas DataFrame objects.
scipy.stats	sp.stats	SciPy provides algorithms and functions for computing problems that arise in science, engineering and statistics. scipy.stats provides the functions for statistics.
statsmodels	sm	statsmodels adds functionality to Python to estimate many different kinds of statistical models, make inferences from those models, and explore data.

12.5 NumPy

The ***NumPy*** (pronounced "Num-pie") package provides tools for mathematical computations in Python. NumPy is used frequently in data science and statistical analysis. NumPy is also frequently used with other data science packages, such as pandas and Matplotlib.

NumPy arrays

The NumPy array data type is called ***ndarray***, where "nd" stands for N-dimensional and N can be any number of dimensions.

- A zero-dimensional array consists of a scalar object. Ex: 2.
- A one-dimensional array consists of a container of scalars. Ex: [2, 4, 6, 8].
- A two-dimensional array consists of a container of containers of scalars. 2D arrays have rows and columns.

An N-dimensional array has N levels of nested containers. At each level, all containers must have the same number of elements. The ***shape***of an array is a tuple of the lengths of each of the array's dimensions. The ***size*** of an array is the total number of elements in an array. Ex: The shape of the 2D array [[2, 4, 6, 8], [12, 14, 16, 18]] is (2, 4) and the size of the array is 8.

Function/Method	Description	Example

`array(object)`	Returns an ndarray based on a given object, like a list.	`# Creates a 1D (1, 4) array based off of a list array1D = np.array([1, 2, 3, 4])`
		
		 `# Creates a 2D (2, 2) array based off of 2 lists array2D = np.array([ [1, 2], [3, 4] ])`

`zeros(arrShape)one(arrayShape)full(arrayShape, value)`	Returns an ndarray of a specified shape filled with zeros, ones, or a specified value.	`# Creates a 2D (2, 2) array filled with 6s# [ [6, 6], [6, 6] ] array_6fill = np.full((2, 2), 6)`

`array[row_index, col_index]`	Returns the element located at indices [row_index, col_index].	`array2D = np.array([ [1, 2], [3, 4] ])`
		
		 `# Returns 3: Element located at second row (index 1), first column (index 0) elem_1_0 = array2D[1, 0]`

`delete(ndarray, index, axis)`	Returns a new ndarray with a row or column deleted from the given ndarray. Deletes the row or column indicated by the index. If axis = 0, delete	`array2D = np.array([ [1, 2], [3, 4] ])`
		
		 `# Returns a new 1D (1x2) array with the second row (index 1, axis 0) ([3,4]) removed# [1, 2] new_a1D = np.delete(array2D, 1, axis=0)`

Function/Method	Description	Example
	row. If axis = 1, delete column.	
`ndarray.sort(axis)`	Sorts an ndarray in place in ascending order along an axis. If axis=None, the array is flattened into a 1D array, and then sorted. If no argument is passed, sorting occurs along the last axis (axis 1 for a 2D array).	`my_array = np.array([2, 4, 1, 3])` `# Sorts a 1D array in place# [1, 2, 3, 4]` `my_array.sort()`
`ndarray.ravel()`	Returns a flattened (1D) version of the given ndarray.	`array_7 = np.array([ [7, 7], [7, 7] ])` `# Returns a new flattened 1D (1, 4) version of a 2D (2, 2) array# [7, 7, 7, 7]` `array_7flat = array_7.ravel()`
`ndarray.reshape(new_shape)`	Returns a new ndarray containing the elements of the given ndarray with a new shape.	`array1D = np.array([1, 2, 3, 4])` `# Returns a new reshaped 2D (2, 2) version of a 2D (1, 4) array# [ [1, 2], [3, 4] ]` `a_reshaped = array1D.reshape((2,2))`
`ndarray.transpose()`	Returns the transpose of an ndarray.	`# Returns a new transposed version of a_reshaped# [[1, 3], [2, 4]]` `array1_transposed = a_reshaped.transpose()`

12.6 pandas

Introduction to pandas and dataframes

pandas is a Python package that stores and manipulates 2-dimensional datasets.

pandas represents datasets with a ***dataframe*** object, of data type ***DataFrame***, which consists of rows and columns. A dataframe's columns contain the features of the dataset and the rows contain the number of instances.

A dataframe's row labels are known as the ***index*** and column labels are known as the ***columns***. Usually, row labels are automatically generated integers, and column labels are manually specified strings. All values in a column must have the same type, but different columns may have different types.

A pandas DataFrame is similar to an NumPy array because both:

- Are indexed, ordered, and mutable containers
- Represent data in multiple dimensions, or axes
- Have a shape attribute, a tuple of integers representing the number of elements along each axis

However, dataframes and arrays also differ in several ways:

- Dataframes are always two-dimensional. Arrays may have zero, one, or many dimensions.
- Different dataframe columns may have different types. All array values have the same type.
- Dataframe labels may be integers, strings, or other types. Array indices are integers only.

Subsetting data involves choosing specific rows and columns from a dataframe according to labels, indices, and slices (a range between 2 indices).

12.7 Matplotlib

Matplotlib is a package used to create static, dynamic, and interactive plots. Seaborn, another common data visualization package used primarily for statistical graphs, is based on Matplotlib.

Figures can be created using the ***pyplot*** library, a state-based interface to the Matplotlib package that uses a syntax similar to MATLAB. Each line of code adds a plot element to the figure one at a time, while preserving previously added elements in the figure.

Other useful functions in the pyplot library include:

- `figure()`: Creates a new figure for a plot to appear in. Ex: `plt.figure(figsize=[4,5])`

- `show()`: Displays the figure and all the objects the figure contains. Ex: `plt.show()`
- `savefig(fname)`: Saves the figure in the current working directory with the filename fname. Ex: `plt.savefig(line_plot.png)`

Pandas Notes

The command `import pandas as pd` imports the pandas modules and shortens the module to 'pd.'

.loc[ ] method

1. Single label:

```
df.loc['row_label'] # Selects one row or one cell.
```

2. List of labels:

```
df.loc[['row1_label', 'row2_label']] # Selects multiple row or c
```

3. Slicing:

```
df.loc['row1_label':'row2_label'] # Select a range of labels.
```

4. Boolean array:

```
df.loc[df['column_name'] > 5] # Select rows based on a conditio
```

5. Conditional selection:

```
df.loc['row_label','column_name'] # Combine label-based selecti
```

Integer-based indexing is done using the `.iloc( )` method.

index method

```
df.index # returns a RangeIndex()
```

Setting an index

```
df = df.set_index('column_name') # Set a column as an index.
```

Returning an index

```
df = df.reset_index() # Resets index to integer values.
```

`Series` objects

A `Series` object has three attributes (in the scope of APS106): `index`, `name`, and `data`.

Modifying a column

```
df.loc[:, column_name] # Accesses a specific column,
# which can be re-assigned to a new list or Series.
```

```
df = df.drop(['column1_name', 'column2_name'], axis="columns"]
# Drops the listed columns along the columns axis.
```

Utility Methods (needed for APS106)

Head

```
df.head() # Prints first 5 rows, can place an int argument as we
```

Tail

```
df.tail() # Prints last 5 rows, can place an int argument as wel
```

Maximum

```
df.max() # Returns the maximum value along a specified row/colur
```

Minimum

```
df.min() # Returns the minimum value along a specified row/colur
```

Mean

```
df.mean() # Returns the arithmetic mean along a column.
```

Value counting

```
df.loc[:, 'column_name'].value_counts()
# Counts the number of times each unique value occurs in a Serie
# Output is a series where the index is the unique values.
```

Unique

```
df.loc['column'].unique() # Returns a list of all the unique val
```

Sorting values

```
df.loc[:, 'column'].sort_values() # Sorts column in ascending or

df.sort_values(by='column', ascending='False') # In descending c
```

Changing value types

```
df.loc[:, 'column'].astype(int) # Converts values in column to i
```

String Methods (needed for APS106)

Uppercase/lowercase

```
df.loc[:, 'column'].str.upper() # Uppercases all values in colu
df.loc[:, 'column'].str.lower() # Lowercases all values in colu
```

Length

```
df.loc[:, 'column'].str.len() # Returns length of each string i
```

Starts with and ends with

```
df.loc[:, 'column'].str.startswith() # Returns list of Boolean
```

Replace

```
df['column'] = df['column'].str.replace('original', 'new')
# Replaces every occurence of the original string with the new
```