Contents

Python Object and Data Structure Basics

Numbers and more in Python

In this lecture, we will learn about numbers in Python and how to use them.

We'll learn about the following topics:

- Types of Numbers in Python
  - integers, Floating Literals
- Basic Arithmetic
  - +, - , /, //, *, %
- Differences between classic division and floor division
- Object Assignment in Python
  - Using variable names can be a very useful way to keep track of different variables in Python.

Python Object and Data Structure Basics

Variable Assignment

- Names can not start with a number
- Names can not contain spaces, use underscore (_) instead
- Names can not contain any of these symbols:

  :'",<>/?|\!@#%^&*~-+

- It's considered best practice (PEP8) that names are lowercase with underscores
- Avoid using Python built-in keywords like list and str
- Avoid using the single characters l (lowercase letter el), O (uppercase letter oh) and I (uppercase letter eye) as they can be confused with 1 and 0

Python Object and Data Structure Basics

Strings

- Strings are used in Python to record text information, such as names.
- Strings in Python are actually a sequence, which basically means Python keeps track of every element in the string as a sequence.
- For example, Python understands the string "hello' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).
- In this lecture we'll learn about the following:
  - Creating Strings
  - Printing Strings
  - String Indexing and Slicing
  - String Properties
  - String Methods
  - Print Formatting

Python Object and Data Structure Basics

String Formatting

- String formatting lets you inject items into a string rather than trying to chain items together using commas or string concatenation. As a quick comparison, consider:

```
player = 'Thomas’

points = 33


'Last night, '+player+' scored '+str(points)+' points.'  # concatenation

'Last night, {player} scored {points} points.'          # string formatting
```

- There are three ways to perform string formatting.
  - The oldest method involves placeholders using the modulo % character.
  - An improved technique uses the .format() string method.
  - The newest method, introduced with Python 3.6, uses formatted string literals, called f-strings.

Python Object and Data Structure Basics

Lists

- Earlier when discussing strings we introduced the concept of a sequence in Python.
- Lists can be thought of the most general version of a sequence in Python.
- Unlike strings, they are mutable, meaning the elements inside a list can be changed!
- In this section we will learn about:
  - Creating lists
  - Indexing and Slicing Lists
  - Basic List Methods
  - Nesting Lists
  - Introduction to List Comprehensions
- Lists are constructed with brackets [] and commas separating every element in the list.

Python Object and Data Structure Basics

Dictionaries

- We've been learning about sequences in Python but now we're going to switch gears and learn about mappings in Python. If you're familiar with other languages you can think of these Dictionaries as hash tables.
- This section will serve as a brief introduction to dictionaries and consist of:
  - Constructing a Dictionary
  - Accessing objects from a dictionary
  - Nesting Dictionaries
  - Basic Dictionary Methods
- **Mappings**: Mappings are a collection of objects that are stored by a key, unlike a sequence that stored objects by their relative position. This is an important distinction, since mappings won't retain order since they have objects defined by a key.
- A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

Python Object and Data Structure Basics
Tuples

- In Python tuples are very similar to lists, however, unlike lists they are immutable meaning they can not be changed.
- You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.
- In this section, we will get a brief overview of the following:
  - Constructing Tuples
  - Basic Tuple Methods
  - Immutability
  - When to Use Tuples
- You'll have an intuition of how to use tuples based on what you've learned about lists. We can treat them very similarly with the major distinction being that tuples are immutable.
- The construction of a tuples use () with elements separated by commas.

Python Object and Data Structure Basics
Sets and Booleans

Sets:

- Sets are an unordered collection of *unique* elements.
- We can construct them by using the set() function.

  Example: x = set()

  x.add(1)

  Output: {1}

- Note the curly brackets. This does not indicate a dictionary! Although you can draw analogies as a set being a dictionary with only keys.

Boolean:

- Python comes with Booleans (with predefined True and False displays that are basically just the integers 1 and 0).
- It also has a placeholder object called None.
- We can use None as a placeholder for an object that we don't want to reassign yet.
- We can also use comparison operators to create Booleans.

Python Object and Data Structure Basics
Files

- Python uses file objects to interact with external files on your computer.
- These file objects can be any sort of file you have on your computer, whether it be an audio file, a text file, emails, Excel documents, etc.
- **Note**: You will probably need to install certain libraries or modules to interact with those various file types, but they are easily available.
- Python has a built-in open function that allows us to open and play with basic file types.
- Various functions we can perform on files are,
  - IPython writing a file
  - Opening a file
  - Writing to a file
  - Appending a file (with %%writefile)
  - Iterating through a file

Contents

Python Comparison Operators

Comparison Operators

- Comparison Operators in Python will allow us to compare variables and output a Boolean value (True or False).
- Below are the various operators which are commonly used.

Operator	Symbol	Description	Example
Equal	==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
Not Equal	!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true
Greater than	>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
Less than	<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
Greater than or Equal	>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
Less than or Equal	<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Comparison Operators
Chained Comparison Operators

- An interesting feature of Python is the ability to *chain* multiple comparisons to perform a more complex test. You can use these chained comparisons as shorthand for larger Boolean Expressions.

- 1 < 2 < 3

The above statement checks if 1 was less than 2 **and** if 2 was less than 3.

- The **and** is used to make sure two checks have to be true in order for the total check to be true.

The above statement can be written as - 1<2 and 2<3

- We can also use **or** to write comparisons in Python. With the **or** operator, we only need one *or* the other to be true

The above statement can be written as – 1<2 or 2<3

Contents

Python Statements

Introduction to Python Statements

- There are two reasons we take this approach for learning the context of Python Statements:
  - If you are coming from a different language this will rapidly accelerate your understanding of Python.
  - Learning about statements will allow you to be able to read other languages more easily in the future.

Python Vs Other Languages:

```
if x:
    if y:
        code-statement
else:
    another-code-statement
```

- Python is so heavily driven by code indentation and whitespace.
- Code readability is a core part of the design of the Python language.

Python Statements

if, elif, else Statements

- **if** Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

Example: "Hey if this case happens, perform some action"

- We can then expand the idea further with elif and else statements

Example: "Hey if this case happens, perform some action. Else, if another case happens, perform some other action. Else, if *none* of the above cases happened, perform this action.‟

- Syntax:

```
if case1:
    perform action1
elif case2:
    perform action2
else:
    perform action3
```

Python Statements
For Loop, While Loop

For loop: A for loop acts as an iterator in Python. It goes through items that are in a sequence or any other iterable item.

- Objects that we've learned about that we can iterate over include strings, lists, tuples, and even built-in iterables for dictionaries, such as keys or values.
- The general format for a for loop in Python is as follows

```
for item in object:
    statements to do stuff
```

While Loop: A while statement will repeatedly execute a single statement or group of statements as long as the condition is true.

- The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.
- The general format of a while loop is:

```
while test:
    code statements
else:
    final code statements
```

Python Statements
break, continue, pass

- We can use break, continue, and pass statements in our loops to add additional functionality for various cases. The three statements are defined by:
  - **break:** Breaks out of the current closest enclosing loop.
  - **continue:** Goes to the top of the closest enclosing loop.
  - **pass:** Does nothing at all.
- Thinking about break and continue statements, the general format of the while loop looks like this:

```
while test:
   code statement
if test:
   break
if test:
   continue
else:
```

- break and continue statements can appear anywhere inside the loop's body, but we will usually put them further nested in conjunction with an if statement to perform an action based on some condition.

Python Statements

Useful Operators

- There are a few built-in functions and "operators" in Python that don't fit well into any category
- **range():** The range function allows you to quickly generate a list of integers.
- **Enumerate():** Enumerate is a very useful function to use with for loops. It keeps track of how many loops are done and automatically creates and updates the index_count or loop_count variable.
- **Zip():** You can use the zip() function to quickly create a list of tuples by "zipping" up together two lists.
- **In()** operator: We've already seen the in keyword during the for loop
- **Min() and max():** We can check the minimum or maximum of a list with these functions.
- **Random():** Python comes with a built in random library. There are a lot of functions included in this random library

  Example: randint(min_range_value, max_range_value)
- **Input()**

Python Statements
List Comprehensions

- In addition to sequence operations and list methods, Python includes a more advanced operation called a list comprehension.
- List comprehensions allow us to build out lists using a different notation. You can think of it as essentially a one line for loop built inside of brackets.
- **Example:** # Square numbers in range and turn into list

```
lst = [x**2 for x in range(0,11)]
lst
```

Output: `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]`

- We can also perform nested list comprehensions.
- **Example:**

```
lst = [ x**2 for x in [x**2 for x in range(11)]]
lst
```

Output: `[0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000]`

Contents

Methods and Functions

Methods

- Methods are essentially functions built into objects.
- Later on in the course we will learn about how to create our own objects and methods using Object Oriented Programming (OOP) and classes.
- Methods perform specific actions on an object and can also take arguments, just like a function.
- The general syntax for methods is,

  object.method(arg1,arg2,etc…)

- with iPython and the Jupyter Notebook we can quickly see all the possible methods using the tab key. Below are the few methods for a list.
  - Append()
  - Count()
  - Extend()
  - Insert()
  - Pop()
  - Remove()
  - Reverse()
  - Sort()
- You can always use Shift+Tab in the Jupyter Notebook to get more help about the method. In general Python you can use the help() function

Methods and Functions

Functions

- A function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.
- Functions will be one of most basic levels of reusing code in Python, and it will also allow us to start thinking of program design

Def Statements:

- We begin with def then a space followed by the name of the function. A pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function followed by a colon.
- We must indent to begin the code inside your function correctly. Python makes use of *whitespace* to organize code.
- The docstring is where we write a basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions.

```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (docstring) goes
    '''
    # Do stuff here
    # Return desired result
```

Return(): Allows a function to return a result that can then be stored as a variable.

Methods and Functions

Lambda Expressions, map and filter

- **MAP():** The **map** function allows you to "map" a function to an iterable object. That is to say you can quickly call the same function to every item in an iterable, such as a list.
- Syntax: **map(<function_name>,<function arguments>)**
- **FILTER():** The filter function returns an iterator yielding those items of iterable for which function(item) is true.
- We need to filter by a function that returns either True or False.
- Syntax: **filter(<function_name>,<function arguments>)**
- **Lambda Expression:** Lambda expressions allow us to create "anonymous" functions. Which basically means we can quickly make ad-hoc functions without needing to properly define a function using def.
- Lambda's body is a single expression, not a block of statements.
- Lambda is designed for coding simple functions, and def handles the larger tasks.
- Example: A lambda expression can be written as follows.

```
lambda num: num ** 2
```

- You will find yourself using lambda expressions often with certain non-built-in libraries, for example the pandas library for data analysis works very well with lambda expressions.

Methods and Functions

Nested Statements and Scope

- When you create a variable name in Python the name is stored in a *name-space*.
- Variable names also have a *scope*, the scope determines the visibility of that variable name to other parts of your code.
- The idea of scope can be described by 3 general rules:

1. Name assignments will create or change local names by default.
2. Name references search (at most) four scopes, these are: Local, Enclosing functions, Global and Built-in.
3. Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes.

- The statement in #2 above can be defined by the **LEGB rule**.

L: Local — Names assigned in any way within a function (def or lambda), and not declared global in that function.

E: Enclosing function locals — Names in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) — Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) — Names preassigned in the built-in names module : open, range, SyntaxError,...

Methods and Functions

*args and **kwargs

- ***args:**
- When a function parameter starts with an asterisk, it allows for an arbitrary number of arguments, and the function takes them in as a tuple of values.

  Syntax: **def myfunc(*args):**

- ****kwargs**
- Python offers a way to handle arbitrary numbers of keyworded arguments. Instead of creating a tuple of values, **kwargs builds a dictionary of key/value pairs.

  Syntax: **def myfunc(**kwargs):**

- We can pass *args and **kwargs into the same function, but *args have to appear before **kwargs

  Syntax: **def myfunc(*args, **kwargs):**

Contents

Object Oriented Programming

Objects, Class, Methods, Attributes

- **Objects:** In Python, *everything is an object*. We can use type() to check the type of object something is.
- **Class:** User defined objects are created using the class keyword.
- The class is a blueprint that defines the nature of a future object.
- From classes we can construct instances. An instance is a specific object created from a particular class.
- **Attributes:** An **attribute** is a characteristic of an object. The syntax for creating an attribute is:

  self.attribute = something
- Each attribute in a class definition begins with a reference to the instance object.
- **Method:** Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects.
- __init__() method is used to initialize the attributes of an object.

Object Oriented Programming

Inheritance, Polymorphism, Special Methods

Inheritance: Inheritance is a way to form new classes using classes that have already been defined.

- The newly formed classes are called derived classes, the classes that we derive from are called base classes.
- The derived classes (descendants) override or extend the functionality of base classes (ancestors).
- Important benefits of inheritance are code reuse and reduction of complexity of a program.

Polymorphism: In Python, *polymorphism* refers to the way in which different object classes can share the same method name, and those methods can be called from the same place even though a variety of different objects might be passed in.

- Real life examples of polymorphism include:
  - opening different file types - different tools are needed to display Word, pdf and Excel files
  - adding different objects - the + operator performs arithmetic and concatenation

Special Methods: These special methods are defined by their use of underscores. They allow us to use Python specific functions on objects created through class.

  - __init__(), __str__(), __len__() and __del__() methods are special methods.

Contents

Modules and Packages

Modules, built-in modules and writing modules

Modules: Modules in Python are simply Python files with the .py extension, which implement a set of functions.

- Modules are imported from other modules using the import command.
- To import a module, we use the **import** command.
- The full list of built-in modules in the Python standard library can be referred in here.
- Two very important functions come in handy when exploring modules in Python - the dir and help functions.
  - We can look for which functions are implemented in each module by using the **dir()** function
  - When we find the function in the module we want to use, we can read about it more using the **help()** function, inside the Python interpreter

Writing Modules: To create a module of your own, simply create a new .py file with the module name, and then import it using the Python file name (without the .py extension) using the import command.

Modules and Packages

Writing Packages

Writing Packages: Packages are name-spaces which contain multiple packages and modules themselves.

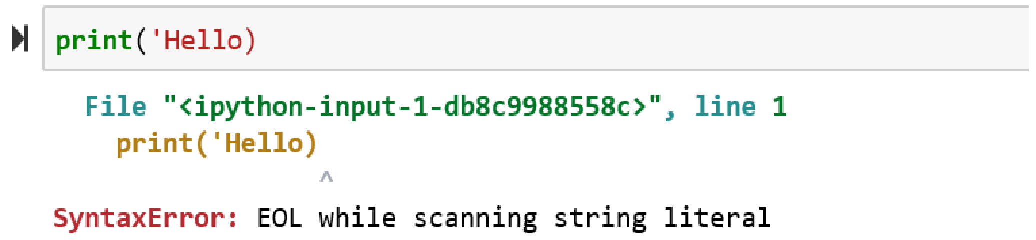
- Each package in Python is a directory which MUST contain a special file called ***_init_*.py**.
- This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.
- If we create a directory called foo, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the ***_init_*.py** file inside the foo directory.
- To use the module bar, we can import it in two ways:
  - import foo.bar (In this method, we must use the foo prefix whenever we access the module bar)
  - from foo import bar
- The ***_init_*.py** file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the ***_all_*** variable

Contents

Errors and Exception Handling

Errors, Exceptions

Error:



```
print('Hello)
```

```
  File "<ipython-input-1-db8c9988558c>", line 1
    print('Hello)
                 ^
SyntaxError: EOL while scanning string literal
```

- Note how we get a SyntaxError, with the further description that it was an EOL (End of Line Error) while scanning the string literal.
- This is specific enough for us to see that we forgot a single quote at the end of the line. Understanding these various error types will help you debug your code much faster.

Exception: The type of error and description is known as an Exception.

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
- Errors detected during execution are called exceptions and are not unconditionally fatal.
- You can check out the full list of built-in exceptions here.

Errors and Exception Handling

Exception Handling – try, except, finally

- **try: and except:** The basic terminology and syntax used to handle errors in Python are the try and except statements. The code which can cause an exception to occur is put in the try block and the handling of the exception is then implemented in the except block of code.

```
try:
   You do your operations here…
   …
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   …
else:
   If there is no exception then execute this block.
```

- **finally:** The finally: block of code will always be run regardless if there was an exception in the try code block. The syntax is:

```
try:
   Code block here
   …
   Due to any exception, this code may be skipped!
finally:
   This code block would always be executed.
```

Errors and Exception Handling

Unit Testing

- Equally important as writing good code is writing good tests.

Testing Tools: There are dozens of good testing libraries out there. Most are third-party packages that require an install, such as:

  - Pylint,
  - pyflakes
  - Pep8

- These are simple tools that merely look at your code, and they'll tell you if there are style issues or simple problems like variable names being called before assignment.
- A far better way to test your code is to write tests that send sample data to your program, and compare what's returned to a desired outcome.
- Two such tools are available from the standard library:
  - Unittest
  - doctest

Errors and Exception Handling

Testing tools - Pylint, Unittest

Pylint:

- **pylint** tests for style as well as some very basic program logic.
- **pylint** tells us if there's any unused variable, but it doesn't know that we might get an unexpected output!
- To install pylint, below is the syntax.

```
! pip install pylint
```

- To test any file, below is the syntax.

```
! pylint <filename>.py
```

Unittest:

- unittest lets you write your own test programs.
- The goal is to send a specific set of data to your program, and analyze the returned results against an expected result.

Contents

Python Built-in Functions

Map()

map():

- Map() is a built-in Python function that takes in two or more arguments: a function and one or more iterables, in the form:

  map(function, iterable, ...)

- map() returns an *iterator* - that is, map() returns a special object that yields one result at a time as needed.

map() with multiple iterables:

- map() can accept more than one iterable.
- The iterables should be the same length
- In the event that they are not, map() will stop as soon as the shortest iterable is exhausted.

Python Built-in Functions

Reduce()

Reduce():

- The function reduce() continually applies the function to the sequence. It then returns a single value.
- Synatx: **reduce(function, sequence)**
- If seq = [s1, s2, s3, ... , sn] is calling reduce(function, sequence), it works like this:
  - At first the first two elements of seq will be applied to function, i.e. func(s1,s2)
  - The list on which reduce() works looks now like this: [function(s1, s2), s3, ... , sn]
  - In the next step the function will be applied on the previous result and the third element of the list, i.e. function(function(s1, s2),s3)
  - The list looks like this now: [function(function(s1, s2),s3), ... , sn]
  - It continues like this until just one element is left and return this element as the result of reduce()

Python Built-in Functions

Filter()

Filter():

- The function filter() offers a convenient way to filter out all the elements of an iterable, for which the function returns True.

  Syntax: **filter(function, list)**

- The function filter() needs a function as its first argument.
- The function needs to return a Boolean value (either True or False).
- This function will be applied to every element of the iterable.
- Only if the function returns True will the element of the iterable be included in the result.
- Like map(), filter() returns an *iterator* - that is, filter yields one result at a time as needed.
- filter() is more commonly used with lambda functions, because we usually use filter for a quick job where we don't want to write an entire function.

Python Built-in Functions

Zip()

- **zip()** makes an iterator that aggregates elements from each of the iterables.
- Returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables.
- The iterator stops when the shortest input iterable is exhausted.
- With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.
- zip() is equivalent to:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

- zip() should only be used with unequal length inputs when you don’t care about trailing, unmatched values from the longer iterables.

Python Built-in Functions

Enumerate()

Enumerate():

- Enumerate allows you to keep a count as you iterate through an object.
- It does this by returning a tuple in the form (count,element).
- The function itself is equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

- enumerate() becomes particularly useful when you have a case where you need to have some sort of tracker
- It takes an optional "start" argument to override the default value of zero

Python Built-in Functions

all(), any()

- all() and any() are built-in functions in Python that allow us to conveniently check for Boolean matching in an iterable.

all() will return True if all elements in an iterable are True.

- It is the same as this function code:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

any() will return True if any of the elements in the iterable are True.

- It is equivalent to the following function code:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

Python Built-in Functions

complex()

COMPLEX():

- complex() returns a complex number with the value real + imag*1j or converts a string or number to a complex number.
- f the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter.
- The second parameter can never be a string.
- Each argument may be any numeric type (including complex).
- If imag is omitted, it defaults to zero and the constructor serves as a numeric conversion like int and float.
- If both arguments are omitted, returns 0j.
- If you are doing math or engineering that requires complex numbers (such as dynamics, control systems, or impedance of a circuit) this is a useful tool to have in Python.

Contents

Python Generators

Iterators, Generators, next(), iter()

Generators:

- Generator functions allow us to write a function that can send back a value and then later resume to pick up where it left off.
- The main difference in syntax will be the use of a **yield** statement.
- The difference between a generator function and a normal function is when a generator function is compiled they become an object that supports an iteration protocol.
- The generator computes one value and then suspends its activity awaiting the next instruction. This feature is known as ***state suspension*.**
- Generators are best for calculating large sets of results (particularly in calculations that involve loops themselves) in cases where we don't want to allocate the memory for all of the results at the same time.

next() and iter():

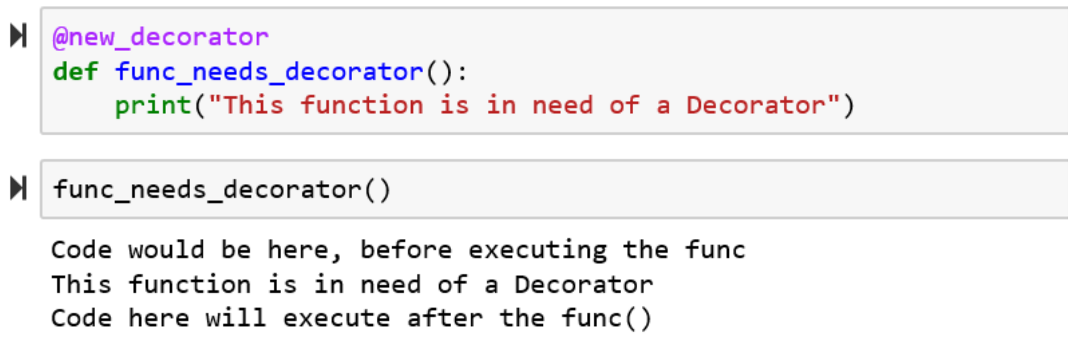
- The **next()** function allows us to access the next element in a sequence.
- After yielding all the values next() caused a StopIteration error. What this error informs us of is that all the values have been yielded.
- A for loop automatically catches this error and stops calling next().
- A string object supports iteration, but we can not directly iterate over it as we could with a generator function. The **iter()** function allows us to do just that.

Contents

Python Decorators

Decorators, writing a decorator

- Decorators can be thought of as functions which modify the *functionality* of another function. They help to make your code shorter and more "Pythonic".
- Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.
- Functions can be defined inside another function and can also be passed as argument to another function.
- Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.
- In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.
- We can rewrite this code using the @ symbol, which is what Python uses for Decorators:



```
@new_decorator
def func_needs_decorator():
    print("This function is in need of a Decorator")
```

```
func_needs_decorator()
```

```
Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()
```