1. Iterable Objects:
   1. Iteration In Python: Iterations and Comprehensions
      1. Iteration Protocol: method call model used by Python
      2. Comprehensions: applies an expression to items in an iterable
      3. for loop can work on any sequence type in Python, including lists, tuples, and strings. for loop turns out to be even more generic than this—it works on any iterable object.
   2. Iterable Objects:
      1. It’s essentially a generalization of the notion of sequences.
      2. An object is said to be iterable if it is either physically stored sequence in memory or an object that generates one item at a time in the context of an iteration operation.
      3. Both types of objects are considered iterable because they support the iteration protocol—they respond to the iter call with an object that advances in response to next calls and raises an exception when finished producing values.
      4. Any object with a \_\_next\_\_ method to advance to a next result, which raises StopIteration at the end of the series of results, is considered iterable in Python.
      5. Any such object may also be stepped through with a for loop or other iteration tool, because all iteration tools normally work internally by calling \_\_next\_\_ on each iteration and catching the StopIteration exception to determine when to exit.
      6. When the for loop begins, it obtains an iterator from the iterable object by passing it to the iter built-in function; the object returned by iter has the required next method.
         1. l = [1,2,3]
         2. i = iter(l)
         3. i.next()
   3. List Comprehension:
      1. It requires less coding on our part and is likely to run substantially faster. The list comprehension isn’t exactly the same as the for loop state- ment version because it makes a new list object (which might matter if there are multiple references to the original list)
      2. In Python, most people find that a list comprehension simply looks like a backward for loop.
      3. L = [x + 10 for x in L if x > 0]
         1. List comprehensions are written in square brackets because they are ultimately a way to construct a new list. They begin with an arbitrary expression that we make up, which uses a loop variable that we make up (x + 10). That is followed by what you should now recognize as the header of a for loop, which names the loop variable, and an iterable object (for x in L).
         2. To run the expression, Python executes an iteration across L inside the interpreter, assigning x to each item in turn, and collects the results of running the items through the expression on the left side. The result list we get back is exactly what the list com- prehension says—a new list containing x + 10, for every x in L.
         3. Technically speaking, list comprehensions are never really required because we can always build up a list of expression results manually with for loops that append results as we go.
         4. However, list comprehensions are more concise to write, and because this code pattern of building up result lists is so common in Python work, they turn out to be very handy in many contexts. Moreover, list comprehensions can run much faster than manual for loop statements (often roughly twice as fast) because their iterations are performed at C language speed inside the interpreter, rather than with manual Python code; es- pecially for larger data sets, there is a major performance advantage to using them.
         5. In general, they are intended for simple types of iterations; for more involved work, a simpler for statement structure will probably be easier to understand and modify in the future. As usual in programming, if something is difficult for you to understand, it’s probably not a good idea.
2. Looping
   1. while
      1. break: breaks out of the closest enclosing loop
      2. continue: jumps to the top of the closest enclosing loop
      3. pass: does nothing. It is an empty statement placeholder.
      4. else: Runs only if the loop is exited normally.
   2. for
      1. It can step through the items in any ordered sequence or other iterable object. The for statement works on strings, lists, tuples, and other built-in iterables, as well as new user-defined objects.
      2. range: produces a series of successively higher integers, which can be used as indexes in a for
      3. zip: returns a series of parallel-item tuples, which can be used to traverse multiple sequences in a for loop. Built-in zip function allows us to use ‘for’ loops to visit multiple sequences in parallel—not overlapping in time, but during the same loop. In basic operation, zip takes one or more sequences as arguments and returns a series of tuples that pair up parallel items taken from those sequences. zip truncates result tuples at the length of the shortest sequence when the argument lengths differ.
      4. **enumerate**: generates both the values and indexes of items in an iterable, so we don’t need to count manually.
      5. map: takes a function and one or more sequence arguments and collects the results of calling the function with parallel items taken from the sequence(s).
      6. filter selects items for which a function is true
      7. reduce runs pairs of items in an iterable through a function.
      8. for target in object: # Assign object items to target

statements # Repeated loop body: use target

if test: break # Exit loop now, skip else

if test: continue # Go to top of loop now

else: # Optional else part

statements # If we didn't hit a 'break'

* + 1. for loops may run quicker than while-based counter loops
    2. As a general rule, use for instead of while whenever possible, and don’t use range calls in for loops except as a last resort
    3. Using range and len together
       1. for loop assignment: using range & len together to split and reorder

S = 'spam' # create spam pams amsp msp

for i in range(len(S)):

X = S[i:] + S[:i]

print(X, end=' ')

# spam pams amsp mspa

* + - 1. for loop assignment: # print every second character in list
         1. Using range & len:

for i in range(0, len(S), 2): print(S[i], end=' ')

* + - * 1. To visit every second character in S, for example, slice with a stride of 2:
        2. a = [1,2,3,4,5,6,7,8]

for i in a[::2]: print(i)

1. Functions
   1. A function is a device that groups a set of statements so they can be run more than once in a program. Functions also can compute a result value and let us specify parameters that serve as function inputs, which may differ each time the code is run.
   2. Functions are the most basic program structure Python provides for maximizing code reuse and minimizing code redundancy.
   3. Arguments in function:
      1. Assign non keyword arguments by position.
      2. Assign keyword arguments by matching names.
      3. Assign extra non keyword arguments to \*name tuple.
      4. Assign extra keyword arguments to \*\*name dictionary.
      5. Assign default values to unassigned arguments in the header.

| Syntax | Location | Interpretation |
| --- | --- | --- |
| func(value) | Caller | Normal argument: matched by position |
| **func(name=value)** | Caller | Keyword argument: matched by name |
| func(\*iterable) | Caller | Pass all objects in iterable as individual positional arguments |
| **func(\*\*dict)** | Caller | Pass all key/value pairs in dict as individual keyword arguments |
| def func(name) | Function | Normal argument: matches any passed value by position or name |
| def func(name=value) | Function | Default argument value, if not passed in the call |
| def func(\*name) | Function | Matches and collects remaining positional arguments in a tuple |
| def func(\*\*name) | Function | Matches and collects remaining keyword arguments in a dictionary |
| def func(\*other, name) | Function | Arguments that must be passed by keyword only in calls (3.X) |
| def func(\*, name=value) | Function | Arguments that must be passed by keyword only in calls (3.X) |

Statements related to functions:

| Statement | Examples |
| --- | --- |
| Calls | myfunc('spam', 'eggs', meat=ham) |
| **def, return** | def adder(a, b=1, \*c):  return a + b + c[0] |
| **global** | def changer():  global x; x = 'new' |
| nonlocal | def changer():  nonlocal x; x = 'new' |
| yield | def squares(x):  for i in range(x): yield i \*\* 2 |
| **lambda** | funcs = [lambda x: x\*\*2, lambda x: x\*3] |
|  |  |

1. Lambda Functions:
   1. lambda argument1, argument2,... argumentN :expression using arguments
   2. Advantages of using Lambda:
      1. lambda is an expression, not a statement. Because of this, a lambda can appear in places a def is not allowed by Python’s syntax
      2. lambda’s body is a single expression, not a block of statements. The lambda’s body is similar to what you’d put in a def body’s return statement; you simply type the result as a naked expression, instead of explicitly returning it. lambda is designed for coding simple functions.
      3. Why Use lambda? Generally speaking, lambdas come in handy as a sort of function shorthand that allows you to embed a function’s definition within the code that uses it. They are entirely optional (you can always use defs instead), but they tend to be simpler coding constructs in scenarios where you just need to embed small bits of executable code.
2. Recursive Functions: **Advanced**
   1. functions that call themselves either directly or indirectly in order to loop
   2. it allows programs to traverse structures that have arbitrary and unpredictable shapes and depths—planning travel routes, analyzing language, and crawling links on the Web. Simple looping statements won’t work here because this is not a linear iteration.
   3. Trees/Backtracking - Computer Science - sorting + tree traversing

# file sumtree.py

def sumtree(L): tot = 0

for x in L:

if not isinstance(x, list):

tot += x

else:

tot += sumtree(x)

return tot

L= [1, [2, [3, 4], 5], 6, [7, 8]]

print(sumtree(L))

1. Scopes and Namespace
   1. Python’s scopes—the places where variables are defined and looked up.
   2. When you use a name in a program, Python creates, changes, or looks up the name in what is known as a namespace—a place where names live. When we talk about the search for a name’s value in relation to code, the term scope refers to a namespace: that is, the location of a name’s assignment in your code determines the scope of the name’s visibility to your code.
   3. By default, all names assigned inside a function are associated with that function’s namespace, and no other. This means that:
      1. Names defined inside a def can only be seen by the code within that def. You cannot even refer to such names from outside the function.
      2. Names defined inside a def do not clash with variables outside the def, even if the same names are used elsewhere. A name X assigned outside a given def (i.e., in a different def or at the top level of a module file) is a completely different variable from a name X assigned inside that def.
   4. To put it concisely:
      1. A namespace is a dictionary, mapping names (as strings) to values. When you do an assignment, like a = 1, you're mutating a namespace. When you make a reference, like print(a), Python looks through a list of namespaces to try and find one with the name as a key.
      2. A scope defines which namespaces will be looked in and in what order. The scope of any reference always starts in the local namespace, and moves outwards until it reaches the module's global namespace, before moving on to the builtins (the namespace that references Python's predefined functions and constants, like range and getattr), which is the end of the line.
      3. Imagine you have a function named inner, nested within a global function named outer, and inner contains a reference to a name. Python first looks in the inner namespace. If the name's not there, Python then looks in the outer namespace. If that fails, Python tries the module's global namespace, then the builtin namespace, eventually throwing a NameError if the name isn't found.
      4. When we say x is in a function's namespace, we mean it is defined there, locally within the function. When we say x is in the function's scope, we mean x is either in the function's namespace or in any of the outer namespaces that the function's namespace is nested inside.
   5. Name Resolution: The LEGB Rule: Lexical Scoping
      1. We call this lexical scoping because variable scopes are determined entirely by the locations of the variables in the source code of your program files.
      2. Local: If a variable is assigned inside a def, it is local to that function.
         1. Y and Z are local to the function (and exist only while the function runs) because they are both assigned values in the function definition: Z by virtue of the = statement, and Y because arguments are always passed by assignment.
      3. Enclosing function: If a variable is assigned in an enclosing def, it is nonlocal to nested functions.
         1. def func1():

x = 90

def func2():

x = 80

* + 1. Global: If a variable is assigned outside all defs, it is global to the entire file.
       1. Global names: X, func
       2. X is global because it’s assigned at the top level of the module file; it can be refer- enced inside the function without being declared global. func is global for the same reason; the def statement assigns a function object to the name func at the top level of the module.
    2. Built-in: Names preassigned in the built-in names module: open, list, return
  1. The whole point behind this name-segregation scheme is that local variables serve as temporary names that you need only while a function is running. For instance, in the preceding example, the argument Y and the addition result Z exist only inside the function; these names don’t interfere with the enclosing module’s namespace (or any other function, for that matter).
  2. Comprehension variables—the variable X used to refer to the current iteration item in a comprehension expression such as [X for X in I]. Because they might clash with other names and reflect internal state in generators, in 3.X, such variables are local to the expression itself in all comprehension forms: generator, list, set, and dictionary.
  3. Exception variables—the variable X used to reference the raised exception in a try statement handler clause such as except Exception as e. Because they might defer garbage collection’s memory recovery, in 3.X, such variables are local to that except block, and in fact are removed when the block is exited.

**class**