ClosingTheDivide
Official Document
Program Structures
San Francisco, California

Program Structures

Zakaria Kortam

Table of Contents

The contents of this document have taken inspiration from Composing Programs by John DeNero.
http://composingprograms.com/

Chapter 1 - Functional Programming

A programming language serves as a framework for organizing ideas about computational processes and facilitates communication among programmers. Powerful programming languages have three mechanisms: primitive expressions and statements, means of combination to build compound elements, and means of abstraction to name and manipulate compound elements.In programming, we work with functions and data. Functions describe rules for manipulating data, and a powerful programming language should be able to describe both primitive data and functions, as well as provide methods for combining and abstracting them.

Expressions in programming languages can represent numbers and can be combined with mathematical operators. Call expressions are important compound expressions that apply functions to arguments.Importing modules in Python allows access to functions and other elements provided by those modules. The math module provides mathematical functions, and the operator module provides functions corresponding to infix operators.Names in a programming language are used to refer to computational objects. Assignments bind names to values. The evaluation of expressions involves recursive steps, and the values of subexpressions are combined to evaluate the entire expression. Evaluation depends on the environment, which provides the context for interpreting symbols in expressions.

There are two types of functions: pure functions and non-pure functions. Pure functions have input and output and do not have side effects. Non-pure functions, like the print function, can have side effects, such as generating output. The print function returns None and should not be used in assignment statements. In Python, function definitions allow us to create user-defined functions by associating a name with a compound operation. The syntax for defining a function is as follows:

```python
def <name>(<formal parameters>):
    return <return expression>
```

The `<name>` is the name of the function, and `<formal parameters>` is a comma-separated list of named parameters that the function takes. The `<return expression>` is an expression that is evaluated and returned whenever the function is called. For example, we can define a function called `square` that squares a number by multiplying it with itself:

```python
def square(x):
    return x * x
```

We can then use the `square` function by calling it with arguments:

```python
result = square(5)
```

User-defined functions can also be used as building blocks to define other functions. For instance, we can define a function called `sum_squares` that calculates the sum of squares of two numbers:

```python
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

The `sum_squares` function calls the `square` function within its body to calculate the squares of `x` and `y` and returns their sum.

User-defined functions are used in the same way as built-in functions, and the definition of a function does not reveal whether it is built-in or user-defined. Functions can be called with arguments, and the arguments are bound to the function's formal parameters within its local frame. The order of frames in the environment affects the value associated with a name during evaluation. Names are evaluated to the value bound to them in the earliest frame of the current environment in which the name is found. The process of evaluating a call expression for a user-defined function involves creating a local frame for the function, binding the arguments to the formal parameters within the local frame, and executing the body of the function in the environment that starts with this frame. Each function application has its own independent local frame. Overall, function definitions provide a powerful abstraction technique in Python, allowing us to create reusable and modular code by encapsulating compound operations within named functions. Good functions have several qualities that make them effective abstractions:

1. Single Responsibility: Each function should have a single job that can be described in a short name and a single line of text. Functions that perform multiple jobs should be divided into separate functions.

2. Don't Repeat Yourself (DRY): Redundant logic should be avoided by implementing it once, giving it a name, and reusing it. If you find yourself copying and pasting code, it's an opportunity for functional abstraction.

3. Generalization: Functions should be defined in a general way. Specific cases can often be implemented as special cases of more general functions. This approach improves code readability, reduces errors, and minimizes code duplication.

Documentation plays an important role in writing good functions:

1. Docstrings: A function's docstring provides documentation describing its purpose, behavior, and arguments. It is conventionally triple-quoted and helps others understand and use the function correctly. Docstrings can be accessed using the `help()` function.

2. Comments: Comments in Python are denoted by the `#` symbol and are meant for human readers. They are ignored by the interpreter and can be used to provide additional explanations or clarifications within the code.

Default argument values in Python can make functions more flexible and easier to use:

1. Default Values: By providing default values for function arguments, those arguments become optional when calling the function. If a default value is not provided, the default value specified in the function definition is used instead. This feature allows functions to have additional arguments without requiring them to be provided in every function call.

Using default argument values and documenting functions effectively can improve code readability, maintainability, and flexibility. Control statements are statements in a programming language that control the flow of execution based on logical comparisons. They differ from expressions in that they have no value and are executed to determine the next action for the interpreter. Statements can include assignment, def, and return statements. They are executed to apply changes to the interpreter state. Expressions can also be executed as statements, but their value is discarded. Compound statements in Python consist of multiple clauses and are

composed of other statements. They typically span multiple lines and start with a one-line header ending in a colon.

Local assignment allows us to define functions with a sequence of operations beyond a single expression. Assignment statements bind a name to a value in the local environment and cannot affect the global frame.Conditional statements in Python use if, elif, and else clauses to control the execution of statements based on the truth value of expressions.

Boolean contexts in Python determine the truth value of expressions in control statements. Python has boolean values (True and False) and comparison operators (>, <, ==, etc.) that return boolean values.

Logical operators (and, or, not) can be used to combine the results of comparisons.

Iteration is a form of repetition that allows the execution of the same statements multiple times. The while statement is used to express iteration in Python.

Testing functions involves verifying their behavior against expected results. Tests can be written as functions that contain sample calls to the function being tested, and assertions are used to verify the expected results.

Assertions are statements that verify expectations and cause an error if the expression being asserted evaluates to a false value.

Doctests are tests that can be placed directly in the docstring of a function and provide a convenient way to document and test the function.

Functions are a way to abstract compound operations and work with higher-level operations. They allow us to define common patterns and use them as named concepts. Functions can accept other functions as arguments or return functions as values, making them higher-order functions.

For example, we can define functions like `sum_naturals`, `sum_cubes`, and `pi_sum` to compute different types of summations. These functions share a common pattern, and we can abstract that pattern by defining a higher-order function called `summation`. `summation` takes as arguments the upper bound `n` and a function `term` that computes the kth term. We can then use `summation` with different `term` functions to compute specific summations.

Functions can also be used as general methods of computation. For example, the `improve` function is a general method for iterative improvement. It takes an `update` function, a `close` function, and an initial `guess` as arguments. The `update` function is repeatedly applied to the `guess` until the `close` function returns `True`. This general method can be used to compute the golden ratio or approximate the square root of a number.

Nested function definitions allow us to define functions inside other functions. These locally defined functions have access to the names in the environment where they are defined (lexical scoping). This enables us to create functions that are specific to a particular context or computation. For example, we can define a `sqrt` function that contains nested functions `sqrt_update` and `sqrt_close` to compute the square root of a number using the `improve` function.

Overall, functions as arguments and higher-order functions provide powerful abstraction mechanisms and greatly enhance the expressive power of a programming language. They allow us to work with common patterns, generalize computations, and create reusable and modular code.

The example provided illustrates various concepts in programming, including function composition, Newton's method for finding function zeros, currying, and lambda expressions.

- Function composition is demonstrated using the `compose1` function, which takes two functions `f` and `g` and returns a new function that applies `f` to the result of applying `g` to its argument.
- Newton's method is explained as an iterative algorithm for finding function zeros. The `newton_update` function is defined to perform the iteration using a function `f` and its derivative `df`. The `find_zero` function combines `newton_update` with a comparison function to find a zero of `f`.
- Computing roots of arbitrary degree `n` is shown using Newton's method. The `nth_root_of_a` function takes a degree `n` and a value `a` and returns the `n`th root of `a`.
- Currying is introduced as a technique to convert a function that takes multiple arguments into a chain of functions that each take a single argument. The `curried_pow` function is defined as an example.
- Lambda expressions are explained as unnamed functions that can be created on the fly. They are particularly useful for defining simple functions inline. The `compose1` function is redefined using a lambda expression.

These concepts demonstrate the power and flexibility of higher-order functions and functional programming paradigms. They allow for concise and expressive code, enabling the creation of reusable and composable abstractions.

Recursive functions are functions that call themselves, either directly or indirectly. They are a way to break down complex problems into simpler subproblems.

For example, let's consider the problem of summing the digits of a natural number. We can use the operators % and // to separate the number into its last digit and all but the last digit. By applying this separation recursively, we can sum the digits of the number.

Here's an implementation of a recursive function `sum_digits` in Python:

```python
def sum_digits(n):
    if n < 10:
        return n
    else:
        all_but_last, last = n // 10, n % 10
        return sum_digits(all_but_last) + last
```

In this function, if the number `n` is less than 10, we simply return `n` because it is a single digit. Otherwise, we split `n` into its last digit (`last`) and all but the last digit (`all_but_last`). We then recursively call `sum_digits` on `all_but_last` and add `last` to the result.

This recursive function applies successfully by breaking down the problem into simpler subproblems. Each recursive call takes a smaller argument until a single-digit input is reached.

Recursive functions often follow a pattern with a base case and one or more recursive calls. The base case handles the simplest inputs, and the recursive calls simplify the problem further. The recursive calls continue until the base case is reached.

Recursive functions can also be mutually recursive, where two or more functions call each other. This can help maintain abstraction within a complicated recursive program.

Printing can be used to visualize the computational process of a recursive function. For example, a function `cascade` can be implemented to print all prefixes of a number from largest to smallest and back to largest.

Tree recursion is another pattern where a function calls itself

Chapter 2 - Object Oriented Programming

Chapter 2 of the book focuses on data representation and manipulation. It explains that every value in Python has a class that determines its type and behavior. The chapter introduces native data types, which are built-in types in Python. These types have literals and built-in functions/operators for manipulation.

The native numeric types in Python include integers (int), real numbers (float), and complex numbers (complex). Integers represent exact values, while floats represent approximations of real numbers with limited precision. This can lead to approximation errors in calculations involving floats. The chapter emphasizes the importance of understanding the differences between int and float objects.

Apart from numeric types, Python also has other native data types for representing various kinds of data, such as sounds, images, web addresses, and network connections. Some of these types are predefined (e.g., bool for True and False), while others can be defined by programmers using combination and abstraction techniques.

The chapter concludes by mentioning that more details and examples about Python's native data types can be found in the online book "Dive Into Python 3," which provides a pragmatic overview and practical tips for manipulating these types.

Data abstraction is a design methodology that involves structuring programs to operate on abstract data. It aims to separate the representation of data from its manipulation, increasing program modularity and making it easier to design, maintain, and modify programs.

In the context of compound data, such as geographic positions or rational numbers, data abstraction allows us to treat the data as a single conceptual unit while also considering its individual parts. This is achieved by defining a concrete representation of the data as well as a set of functions that operate on abstract data using the concrete representation.

For example, when working with rational numbers, we can define a rational number as a pair of two integers: a numerator and a denominator. We can then define functions such as rational, numer, and denom that manipulate rational numbers using this representation. By using these abstraction functions, we can perform operations like addition, multiplication, printing, and equality testing on rational numbers.

The use of data abstraction introduces an abstraction barrier between different parts of the program. The barrier ensures that each part operates at its appropriate level of abstraction and only uses the necessary operations. Violating the abstraction barrier by directly accessing lower-level functions or representation details can make the program harder to maintain and modify.

Abstraction barriers make it easier to change the representation of data without affecting the behavior of the program. As long as the behavior conditions specified by the abstraction are maintained, the program remains correct. Abstraction barriers also allow us to think about data in a more abstract and independent manner, focusing on the behavior and relationships rather than the specific implementation details.

In some cases, data can be represented using higher-order functions instead of traditional data structures like lists. This functional representation of data can fulfill the necessary behavior conditions and provide an alternative way to represent compound data.

Overall, data abstraction is a powerful design technique that promotes modularity, flexibility, and ease of maintenance in programs by separating the representation and manipulation of data.

Sequences are ordered collections of values that have common behaviors. Python includes various types of sequences, with the most important one being lists. Sequences have a finite length and can be accessed using element selection. Lists can be added together and multiplied by integers to combine and replicate the sequences. Lists can also contain other lists, allowing for nested element selection.

Sequence iteration is a common pattern where the elements of a sequence are processed one by one. This can be achieved using a for statement in Python, which iterates over the elements of the sequence. The for statement binds a name to each element in the sequence and executes a suite of code for each element.

Sequence processing involves performing computations on sequences. List comprehensions are a concise way to express sequence processing operations. They allow for evaluating expressions for each element in a sequence and collecting the results. List comprehensions can also include a filter expression to select elements that satisfy a condition.

Aggregation is another pattern in sequence processing, which involves combining all elements into a single value. Functions like sum, min, and max are examples of aggregation functions. Higher-order functions can be used to express sequence processing patterns, where functions are applied to elements or used for aggregation.

Python provides additional behaviors for sequences, including membership testing using the in and not in operators, and slicing to extract contiguous spans of a sequence using the slice notation.

Sequences are a powerful abstraction in computer science and are extensively used in programming to process and manipulate data.

Partition trees are binary trees used to represent the partitions of an integer. Each node in the tree represents a choice made during computation. The left branch contains partitions that include a specified value, while the right branch contains partitions without that value. The leaves of the tree indicate successful partitions.

Here's an example of a Python code that generates a partition tree:

```python
def partition_tree(n, m):
    if n == 0:
        return [True]
    elif n < 0 or m == 0:
        return [False]
    else:
        left = partition_tree(n - m, m)
        right = partition_tree(n, m - 1)
        return [m, left, right]

tree = partition_tree(6, 4)
print(tree)
```

The output will be:

```
[4, [2, [0, [True], [-1, [False], [False]]], [1, [-2, [False], [False]], [-3, [False], [False]]]], [3, [-1, [False], [False]], [2, [-2, [False], [False]], [-3, [False], [False]]]]]
```

The code uses recursion to construct the partition tree. The `partition_tree` function takes two arguments: `n` (the integer to be partitioned) and `m` (the maximum value allowed in the partitions). The function returns a partition tree represented as a nested list.

To print the partitions from the partition tree, another recursive function called `print_parts` can be used. It traverses the tree and constructs each partition as a list. When a True leaf is reached, the partition is printed.

Here's an example of the `print_parts` function:

```python
def print_parts(tree, partition=[]):
    if isinstance(tree, bool):
        if tree:
            print(' + '.join(partition))
    else:
        m, left, right = tree
        print_parts(left, partition + [str(m)])
        print_parts(right, partition)

print_parts(tree)
```

The output will be:

```
4 + 2
4 + 1 + 1
3 + 3
3 + 2 + 1
3 + 1 + 1 + 1
2 + 2 + 2
2 + 2 + 1 + 1
2 + 1 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1 + 1
```

This code recursively traverses the partition tree and prints the partitions by joining the numbers with "+" symbols. The `partition` parameter keeps track of the current partition being constructed.

Slicing can also be applied to the branches of a tree. For example, a common tree transformation called binarization converts a tree into a right-branching binary tree by grouping adjacent branches.

Here's an example of a Python code that performs right binarization on a tree:

```python
def right_binarize(tree):
    if isinstance(tree, list):
        if len(tree) > 2:
            return [tree[0], tree[1:]]
        return [right_binarize(b) for b in tree]
    return tree

tree = [1, 2, 3, 4, 5, 6, 7]
binarized_tree = right_binarize(tree)
print(binarized_tree)
```

The output will be:

```
[1, [2, [3, [4, [5, [6, 7]]]]]]
```

The `right_binarize` function recursively traverses the tree and converts it into a right-branch

In programming, abstraction helps us manage the complexity of large systems, while modularity allows us to divide systems into separate parts that can be developed and maintained independently. One way to achieve modularity is by incorporating mutable data, where a single object can represent something that changes over time. This is a key aspect of object-oriented programming.

Objects combine data values with behavior, representing information and behaving like the things they represent. They have attributes (named values) and methods (function-valued attributes) that define their behavior. Python objects, such as dates, numbers, strings, lists, and ranges, have attributes and methods that allow them to behave in ways appropriate to their values.

Mutable objects, like lists, can be modified by invoking methods on them. They represent values that change over time. Immutable objects, like numbers and strings, cannot be modified once created.

Lists, as mutable objects, can be modified using methods such as append, extend, remove, insert, sort, and reverse. Changes to a list affect the original object, and multiple names can refer to the same list object. Copying a list creates a new list with the same elements.

Identity and equality are important concepts when working with mutable objects. Two objects can be identical (refer to the same object) or equal (have the same contents). Python provides the is and is not operators for identity comparison and the == operator for equality comparison.

List comprehensions allow for the creation of new lists based on existing ones, without modifying the original list. Tuples, on the other hand, are immutable sequences and cannot be modified once created. However, a tuple can contain mutable elements, and its elements can be accessed and manipulated.

Dictionaries are built-in data structures in Python that store key-value pairs. They allow for efficient lookup and retrieval of values based on descriptive keys. Keys can be any immutable objects, such as strings, and each key can have at most one associated value. Dictionaries support various methods for iterating over their contents and can be converted from a list of key-value pairs using the dict constructor function.

Non-local assignment allows functions to have local state, similar to how lists and dictionaries can have changing values. It allows a function to change the value of a name that is outside of its local frame.

In the example given, the function `make_withdraw` returns a function called `withdraw`, which models the process of withdrawing money from a bank account. The `withdraw` function has access to a non-local name `balance`, which represents the current balance of the account.

By using the `nonlocal` statement, the `withdraw` function can change the value of `balance` and return the updated balance after a withdrawal. If the withdrawal amount exceeds the balance, it returns the message 'Insufficient funds'. Each call to `withdraw` can have a different result, depending on the current balance and the withdrawal amount.

The `make_withdraw` function creates a new environment frame with the initial balance value and returns the `withdraw` function. The `nonlocal` statement in `withdraw` indicates that the `balance` name refers to the first frame in which `balance` is already bound, allowing it to change the value of `balance` in that frame.

The use of non-local assignment introduces the concept of local state to functions, allowing them to maintain their own internal state that evolves over successive calls. However, each instance of a function with local state maintains its own state, and the state is inaccessible to other functions. This allows for the creation of abstractions that model real-world objects with changing internal states, such as bank accounts.

It's important to note that non-local assignment has its costs and nuances. It introduces the concept of changing values, as opposed to immutable values in previous examples. Functions with local state behave differently from simple bindings of immutable values. When multiple names are bound to a function with state, changes to the function's state through one name can affect the behavior of the function accessed through another name if they refer to the same function object.

Message passing is a programming paradigm that involves organizing computation by passing messages between functions or objects. It is based on the idea of sending messages to objects and allowing them to handle those messages according to their internal logic.

In the context of the example provided, message passing is used to implement mutation operations for Python lists and dictionaries. The implementation uses dispatch functions that respond to different messages (e.g., 'extend', 'insert') to perform specific operations on the data structure.

For Python lists, the example shows two possible approaches: implementing mutation operations as separate functions that use existing messages like 'pop_first' and 'push_first', or adding additional conditional clauses to the dispatch function to handle each mutation operation directly.

For dictionaries, the example demonstrates a functional implementation using a list of key-value pairs. The dispatch function handles messages like 'getitem' and 'setitem' to retrieve or modify values associated with specific keys.

The use of dispatch dictionaries is also mentioned, where a dictionary is used as a lookup mechanism to handle messages instead of using conditional statements.

Overall, message passing provides a way to encapsulate the logic for different operations within a single function or object, allowing for flexible and modular code organization.

Object-oriented programming (OOP) is a programming paradigm that organizes programs by creating classes, which serve as templates for objects. Objects are instances of classes and encapsulate data (attributes) and behavior (methods). OOP provides a way to abstract complex systems by modeling them as interacting objects.

In Python, classes are defined using the `class` statement, which includes attributes and methods shared among objects of the class. To create an object (instance) of a class, the class is instantiated using the class name followed by parentheses, similar to calling a function.

Instance attributes are specific to each object and can be accessed using dot notation. Methods are functions defined within a class and operate on the object itself. They can access and manipulate the object's attributes using the `self` parameter, which is automatically bound to the object when a method is invoked.

Object methods are invoked using dot notation, and the object itself is passed implicitly as the first argument (bound to `self`). Class attributes, shared across all objects of a class, are defined outside of any method and can be accessed from any instance.

Dot expressions, such as `object.attribute` or `getattr(object, 'attribute')`, are used to access attributes of an object. If the attribute is a method, it is automatically bound to the object when invoked.

Naming conventions in OOP suggest using CapWords for class names and lowercased words separated by underscores for method names. Attributes or methods starting with an underscore are conventionally considered implementation details and should be accessed only within the class itself.

Class attributes, also known as class variables or static variables, are attributes associated with the class itself and shared across all instances. They are created using assignment statements outside of any method in the class.

Overall, object-oriented programming provides a way to organize code by modeling real-world or abstract concepts as interacting objects, promoting encapsulation, modularity, and code reusability.

In object-oriented programming, inheritance is used to represent relationships between different types of classes. It allows a subclass to inherit attributes and behaviors from a base class, while also allowing the subclass to override certain attributes or methods if needed.

For example, a CheckingAccount can be seen as a specialization of an Account. It inherits attributes from the Account class but may have its own unique attributes or behaviors. In Python, inheritance is specified by placing the base class in parentheses after the class name.

Multiple inheritance is also supported in Python, where a subclass can inherit attributes from multiple base classes. However, when there are conflicts in attribute names between the base classes, the resolution of the ambiguity depends on the specific ordering of the base classes.

The object-oriented paradigm promotes the organization of programs by encapsulating different aspects of the program's state in objects and defining the functions that implement the program's logic in classes. It provides a convenient and flexible way to model systems with separate but interacting parts. However, it's important to recognize that not all abstractions are best represented using classes, and functional abstractions may be more suitable in certain cases. Understanding when to use classes versus functions is a crucial design skill in software engineering.

In this section, the implementation of classes and objects is discussed using functions and dictionaries. The purpose is to show that object-oriented programming can be achieved even in languages that don't have built-in support for objects.

Instances are implemented using dispatch dictionaries that respond to "get" and "set" messages for attribute values. Attributes are stored in a local dictionary called "attributes". The implementation also includes bound method values, where a method is bound to an instance by inserting the instance as the first argument.

Classes are also implemented as objects that respond to "get", "set", and "new" messages. The "get" function for classes queries the base class if an attribute is not found. Classes can create new instances and invoke the "__init__" constructor function after instance creation.

The usage of implemented objects is demonstrated with a bank account example. The "make_account_class" function creates the Account class with methods for deposit and withdrawal. An instance of the Account class is created using the "new" message. Attributes and methods of the instance can be accessed using "get" messages, and methods can be called to update the account balance.

Inheritance is also demonstrated by creating a subclass, CheckingAccount, which overrides a subset of attributes from the base class Account. The subclass imposes a withdrawal fee and has a different interest rate. The subclass calls the corresponding method of the base class when necessary.

The implemented object system closely resembles Python's built-in object system, but with some simplifications. Python has additional features and special methods to handle different types of objects correctly.

Object abstraction in programming allows for the efficient construction and use of abstract data representations. It enables the coexistence of multiple representations of abstract data within a program.

A key concept in object abstraction is the use of generic functions, which can accept values of different types. There are three common techniques for implementing generic functions: shared interfaces, type dispatching, and type coercion. These techniques leverage features of the Python object system to support the creation of generic functions.

String conversion is an important aspect of object abstraction. Objects should provide both a human-interpretable text representation and a Python-interpretable expression representation. The `str` constructor returns a human-readable string, while the `repr` function returns a Python expression that evaluates to an equal object. The `repr` function invokes the `__repr__` method on an object, while the `str` constructor invokes the `__str__` method.

Special methods in Python are invoked by the interpreter in specific circumstances. For example, the `__init__` method is automatically called when an object is constructed, and the `__str__` and `__repr__` methods are invoked for printing and displaying values. Special methods allow for customization of object behavior in various contexts.

Multiple representations of data can be accommodated through abstraction barriers. In large programs, it may be necessary to have multiple representations for a data type. For example, complex numbers can be represented in rectangular form (real and imaginary parts) or polar form (magnitude and angle). By defining appropriate methods and interfaces, it is possible to work with multiple representations of data within a program.

Python supports the definition of interfaces through shared attribute names and behavior specifications. Interfaces allow different data types to respond to the same messages in different ways. Attributes can be computed on-the-fly using the `@property` decorator, allowing for the dynamic calculation of attribute values based on other attributes.

Implementing multiple representations of data using interfaces and abstraction barriers provides flexibility and extensibility to software systems. Different representations can coexist within the same program, and new representations can be added easily by defining classes with the required attributes.

By leveraging object abstraction and multiple representations, programmers can design more flexible and adaptable software systems.

Efficiency in representing and processing data refers to the computational resources used, such as time and memory. Measuring exact time and memory requirements is challenging, so efficiency is often characterized by measuring how many times a certain event occurs, such as a function call.

The Fibonacci function discussed is an example of a tree-recursive function that exhibits redundancy and inefficiency in computation. By counting the number of function calls, we can observe that the number of calls grows rapidly.

The space requirement of a tree-recursive function is typically proportional to the maximum depth of the tree. Memory used for inactive environments can be reclaimed, reducing space requirements.

Memoization is a technique used to improve the efficiency of recursive functions that repeat computation. It involves storing the return values for previously computed arguments. Memoization can be implemented as a higher-order function or decorator.

The efficiency of a process can be analyzed by categorizing it based on the order of growth, which expresses how the resource requirements grow as a function of the input size. Theta notation is used to describe the order of growth, indicating upper and lower bounds.

An example of a process analyzed for efficiency is the function count_factors, which counts the number of integers that evenly divide a given number. The time required for this function is analyzed in terms of the number of steps and its order of growth is determined to be Θ(n^0.5).

Another example is the computation of exponentiation, which can be done using linear recursion, linear iteration, or successive squaring. The recursive approach using successive squaring is more efficient, reducing the number of steps required for computation.
The code provided demonstrates the implementation of recursive objects in Python, specifically linked lists and trees.

The "Link" class represents a linked list, where each instance has a "first" attribute representing the first element of the list and a "rest" attribute representing the remaining elements as another linked list. The class provides methods such as "__len__" (to compute the length of the list), "__getitem__" (to access elements by index), and "__repr__" (to convert a linked list to a string expression). It also demonstrates the concept of recursive methods by invoking itself indirectly through these special method names.

The "Tree" class represents a tree data structure, where each instance has a "label" attribute representing the value at the root of the tree and a "branches" attribute representing the subtrees as a sequence of Tree instances. The class provides methods such as "__repr__" (to convert a tree to a string representation) and "is_leaf" (to check if a tree is a leaf node).

The code also includes functions for manipulating linked lists and trees, such as extending a linked list, mapping and filtering elements of a linked list, and constructing partitions of an integer using a tree-like recursive process. It also demonstrates the use of memoization to optimize the computation of Fibonacci numbers using a tree structure.

Additionally, the code introduces the concept of sets in Python, showcasing different implementations of sets. One implementation represents a set as an unordered sequence using the "Link" class, where membership testing and element adjunction are performed recursively. Another implementation represents sets using the built-in Python set type and demonstrates various set operations.

Overall, the code provides examples and explanations of recursive objects and their applications in Python.

The balanced property of a tree is not guaranteed when elements are added in a specific order. However, if elements are added randomly, the tree tends to be balanced on average. To ensure balance, we can define an operation that transforms an arbitrary tree into a balanced tree with the same elements. By performing this transformation periodically, we can maintain balance in our set.

The intersection and union operations on tree-structured sets can be implemented by converting them to ordered lists, performing the operations on the lists, and then converting the result back to a tree structure. The specific details of this implementation are left as an exercise.

Python's built-in set type uses a different internal representation based on hashing, which allows for constant-time membership tests and adjoin operations. This implementation does not support mutable data types but provides an immutable frozenset class for nested sets.

Chapter 3 - Scheme Programming

Chapter 3 introduces the concept of programs as the third fundamental element of programming. A Python program is a collection of text, and its meaningful computation is performed through interpretation. The interpretation process is carried out by an interpreter, which determines the meaning of expressions in the programming language.

Understanding the role of interpreters in programming allows us to see ourselves as language designers rather than just language users. Programming languages vary in their syntax, features, and application domains. Function definition and application are pervasive constructs in general-purpose programming languages, while some powerful languages may lack features like object systems, higher-order functions, and control statements.

The chapter focuses on the design of interpreters and the computational processes they create during program execution. Despite the perceived complexity of designing interpreters for general programming languages, many interpreters share a common elegant structure: two mutually recursive functions. The first function evaluates expressions in environments, while the second function applies functions to arguments. These functions are defined in terms of each other, as applying a function involves evaluating expressions, and evaluating an expression may require applying functions.

In this section, the text introduces a high-level programming language called Scheme, which is a dialect of Lisp. Scheme is a functional programming language that specializes in symbolic computation and uses only expressions (no statements) and immutable values.

Scheme programs consist of expressions, which can be call expressions or special forms. Call expressions are formed by an operator expression followed by zero or more operand sub-expressions, enclosed in parentheses. Scheme uses prefix notation, where the operator comes before the operands. Expressions can be nested and span multiple lines.

Scheme provides special forms like the "if" expression, which has a different evaluation procedure. The "if" expression evaluates a predicate and based on its result, evaluates and returns either the consequent or the alternative expression.

Scheme supports defining values and functions using the "define" special form. Values can be named using "define" followed by the name and the value. Functions (procedures) can be defined using a similar syntax, with the name, formal parameters, and the body of the function.

Scheme has built-in support for compound values such as pairs and lists. Pairs are created using the "cons" function, and elements can be accessed using "car" and "cdr". Recursive lists can be created by nesting pairs. Scheme provides operations for manipulating lists, such as computing the length and accessing elements.

Scheme allows working with symbolic data by quoting expressions using the single quotation mark. Quoted expressions are not evaluated but treated as data objects. Quotation allows working with symbols themselves rather than their values.

The text also mentions Turtle graphics, which is an illustrating environment included in the Scheme implementation used in the book. Turtle graphics is a graphical tool for drawing lines and shapes based on procedures. The turtle has a position and heading on a canvas and can be moved and turned using procedures.

Finally, the text presents an example of a recursive drawing using Scheme's Turtle graphics. It demonstrates how to draw Sierpinski's triangle, a fractal shape, using a recursive procedure.

Exceptions are an important aspect of programming as they allow programmers to handle errors and exceptional circumstances in their programs. When designing a program, it is crucial to anticipate potential errors and take appropriate measures to handle them.

In Python, exceptions are raised when an error or exceptional condition occurs. The Python interpreter automatically raises exceptions when it detects errors in expressions or statements. Additionally, users can manually raise exceptions using the `raise` statement. An exception is an object instance with a class that inherits from the `BaseException` class. By raising an exception, the normal flow of execution is interrupted, and the program returns to an enclosing part that is designated to handle the exception.

When an exception is raised, the current block of code is not executed further unless the exception is handled. If the exception is not handled, the interpreter will either return to the interactive prompt or terminate the program, depending on how Python was executed. A stack backtrace, which describes the nested set of active function calls leading to the exception, is printed by the interpreter. This backtrace helps in identifying the source of the exception.

To handle exceptions, Python provides the `try` statement. A `try` statement consists of multiple clauses, where the first one starts with `try`, and the rest start with `except`. The code in the `try` block is executed, and if an exception occurs during that execution, the corresponding `except` block is executed. Each `except` clause specifies the particular class of exception it can handle. The exception object raised can be assigned to a variable using the `as` keyword in the `except` clause.

Here's the general structure of a `try` statement:

```python
try:
    <try suite>
except <exception class> as <name>:
```

```
    <except suite>
...
```

In the above code, `<try suite>` is always executed, and the `<except suite>` is executed only when an exception of the specified class is raised. Within the `<except suite>`, the exception object is available using the assigned name, but this binding is limited to the `<except suite>`.

Here's an example of handling a `ZeroDivisionError` exception:

```python
try:
    x = 1 / 0
except ZeroDivisionError as e:
    print('Handling a', type(e))
    x = 0
```

In the above code, if a `ZeroDivisionError` occurs while dividing by zero, the exception is caught, and the code inside the `except` block is executed. In this case, the message "Handling a \<class 'ZeroDivisionError'> is printed, and the variable `x` is assigned a value of 0.

Exceptions can also be handled within functions. If an exception occurs within a function called from within the `try` block, control jumps directly to the `except` block of the most recent `try` statement that handles that specific type of exception.

Here's an example demonstrating exception handling within functions:

```python
def invert(x):
    result = 1 / x  # Raises ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return result

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)
```

In the above code, the `invert_safe` function calls the `invert` function, which may raise a `ZeroDivisionError` if the input is 0. The `invert_safe` function handles this exception and returns a string representation of the exception instead of letting it propagate further.

By using exceptions, programmers can separate the logic for error handling from the rest of the program's logic. This separation improves code modularity and makes it easier to manage and reason about

errors. Exceptions also play a vital role when implementing interpreters or handling exceptional conditions in programs.

In this section (3.4) of the textbook, the author discusses the concept of interpreters for languages that are established in terms of other languages. Interpreters play a crucial role in computer programming as they allow us to implement and evaluate new languages. The author begins by introducing a language called Calculator, which is a limited subset of Scheme. They then describe how to develop an interpreter for Scheme as a whole.

The interpreter for the Calculator language is implemented in Python. It takes string lines as input and evaluates them as Calculator expressions, returning the result. If the expression is not well-formed, the interpreter raises an appropriate exception.

The author explains the concept of expression trees, which are used to represent expressions as data structures in the interpreter. They introduce a class called Pair to represent Scheme pairs and lists in Python. The Pair class is similar to the Rlist class introduced earlier in the chapter.

The process of generating expression trees from raw text input is called parsing. Parsing involves lexical analysis (tokenization) and syntactic analysis. The lexical analyzer partitions the input string into tokens, and the syntactic analyzer constructs an expression tree from the sequence of tokens.

The author provides an example of tokenizing a well-formed Calculator expression and explains the iterative process of lexical analysis. Syntactic analysis is implemented using a recursive function called scheme_read, which analyzes a token sequence and constructs an expression tree. The scheme_read function expects its input to be a Buffer instance, which collects tokens spanning multiple lines.

The author emphasizes the importance of informative syntax errors in improving the usability of an interpreter. The SyntaxError exceptions raised during parsing include descriptions of the encountered problems.

The textbook also covers the evaluation of Calculator expressions. The calc_eval function evaluates an expression by recursively evaluating sub-expressions and applying operators. The calc_apply function applies operators to a list of arguments.

The author discusses the role of calc_eval in making proper calls to calc_apply and demonstrates how the interpreter evaluates nested expressions.

A read-eval-print loop (REPL) is a mode of interaction with an interpreter that reads an expression, evaluates it, and prints the result. The author provides an example implementation of a REPL for the Calculator language. The loop reads input from the user, constructs an expression using the scheme_read function, evaluates it using calc_eval, and prints the result. Error handling is also implemented to report errors to the user without exiting the loop.

The author concludes the section by mentioning that the structure of the read-eval-print loop can be generalized and applied to different languages by parameterizing the parsing function, evaluation function, and exception types handled.

Overall, this section provides an overview of implementing interpreters for languages with combination, using the example of the Calculator language and Scheme as a whole. It covers parsing expressions, constructing expression trees, evaluating expressions, and implementing a read-eval-print loop.

The Scheme interpreter is a general programming language that supports abstraction by binding names to values and defining new operations. It consists of a parser, evaluator, and procedure application process.

The parser extends the scheme_reader module to correctly parse dotted lists and quotation in Scheme expressions.

The evaluator, implemented in the scheme_eval function, handles different forms of expressions in Scheme, such as primitives, special forms, and call expressions. It uses recursion and applies evaluation rules specific to each form.

Procedure application is a more general process in Scheme compared to the Calculator language. It involves applying arguments to PrimitiveProcedures or LambdaProcedures. PrimitiveProcedures are implemented in Python as functions, while LambdaProcedures are implemented in Scheme and evaluated in a new environment.

The evaluation process involves mutual recursion between the eval function (scheme_eval) and the apply function (scheme_apply). They work together to evaluate expressions and apply functions, forming the core of the evaluation process.

Environments are represented by the Frame class, which binds symbols to values. Each Frame instance has a dictionary of bindings and a parent frame. The lookup method searches for a symbol's value in the current frame and its parent frames. The define method binds a symbol to a value in the current frame.

The Scheme interpreter acts as a universal machine, interpreting descriptions of other machines (programs) written in Scheme. It bridges the gap between data objects and the programming language itself. Users can enter Scheme expressions into the interpreter, which will evaluate them based on predefined rules.

In dynamic programming languages like Scheme, the distinction between user programs and interpreter data can be blurred. The ability to evaluate data objects as expressions during program execution is a powerful feature that allows for flexible and dynamic programming.

Chapter 4 - Data Processing

In this chapter, we explore techniques for efficiently processing and manipulating sequential data streams, including those with unbounded or infinite size.

In Chapter 2, we introduced the sequence interface in Python, which is implemented by built-in data types like lists and ranges. Now, we extend the concept of sequential data to include collections that can have unbounded or infinite lengths.

Infinite sequences have mathematical examples, such as the positive integers or the Fibonacci numbers. In computational domains, sequential data sets of unbounded length also arise. Examples include the sequence of telephone calls through a cell tower, the sequence of mouse movements made by a computer user, or the sequence of acceleration measurements from sensors on an aircraft. These sequences continue to grow as events occur or data is generated.

An implicit sequence refers to a representation of a sequence where each element is not stored explicitly in the computer's memory. Instead, elements are computed on demand when they are requested. This approach is known as lazy computation.

Iterators are objects that provide sequential access to values, one by one, in a container. In Python, iterators can be obtained by calling the built-in `iter` function on a container object. The `next` function is used to retrieve the next element from an iterator. When there are no more values available, a `StopIteration` exception is raised.

Any value that can produce iterators is called an iterable. In Python, iterables can be passed to the `iter` function. Examples of iterables include sequence values such as strings and tuples, as well as other containers like sets and dictionaries.

Python provides several built-in functions that return iterators when given iterable values as arguments. These functions include `map`, `filter`, `zip`, and `reversed`.

The `for` statement in Python operates on iterators. An iterable object can be used as the expression in the `for` statement, and the statement iterates over the elements of the iterable by calling the iterator's `__next__` method.

Generators are a type of iterator that are returned by generator functions. Generator functions use the `yield` statement to return elements of a series. Generators do not use object attributes to track their progress but control the execution of the generator function itself.

An object is considered iterable if it returns an iterator when its `__iter__` method is invoked. Iterable objects represent data collections, while iterators track progress through sequential data.

In Python, iterators make a single pass over the elements of a series. To iterate over elements multiple times, generator functions can be used with the `yield` statement.

Overall, iterators and iterables provide a way to process elements of a sequence sequentially, with lazy computation and on-demand element generation being key advantages.

Declarative programming involves describing the desired result of a computation rather than directly specifying the steps of the computation. SQL (Structured Query Language) is a widely used declarative programming language for interacting with databases. In SQL, data is stored in tables consisting of rows (records) with similar structures. Queries are used to retrieve and transform data from these tables.

A select statement in SQL defines a new table by projecting an existing table using a from clause. The resulting table is described by a comma-separated list of expressions, which are evaluated for each row of the input table. The select statement can also include a where clause to filter rows based on a condition and an order clause to specify the ordering of the resulting table.

Joins are used to combine multiple tables in a database. Tables are joined by listing their names separated by commas in the from clause. The resulting table contains a new row for each combination of rows from the input tables. Joins are often accompanied by a where clause to express a relationship between the tables being joined.

To interpret SQL statements, a representation for tables, a parser for SQL statements, and an evaluator for parsed statements are needed. Tables can be represented using classes, and rows can be represented as instances of these classes. Select statements can be executed by joining input tables, filtering and ordering rows, and projecting the resulting rows into columns.

Overall, declarative programming in SQL provides a high-level abstraction for working with databases, allowing users to focus on the desired result rather than the procedural details of computation.

The logic programming section introduces a declarative query language called logic, which is based on Prolog and the declarative language in Structure and Interpretation of Computer Programs. In this language, data records are expressed as Scheme lists, and queries are expressed as Scheme values. The logic interpreter, built on the Scheme project from the previous chapter, retrieves facts from a database and deduces new facts using logical inference.

Facts in the logic language represent records in the database and are declared using the keyword "fact." A fact statement consists of one or more lists, and each fact is a relation that is matched to queries.

Queries, on the other hand, start with the keyword "query" and also consist of one or more lists. Queries can contain variables, which are symbols starting with a question mark. Variables in queries are matched to facts by the query interpreter.

The logic language supports compound facts, where facts can have multiple sub-expressions and variables. A conclusion followed by hypotheses represents a multi-expression fact. Queries can refer to compound facts, and the query interpreter combines the available facts to match the query.

Negation is supported in the logic language using the "not" keyword. A query with negation succeeds if the specified relation fails and fails if the relation succeeds. However, the concept of negation as failure in logic programming might be counterintuitive in some cases.

Recursive facts are also possible in the logic language, where the conclusion of a fact depends on a hypothesis containing the same symbols. Recursive facts allow for chains of inference to match queries to existing facts in the database.

Hierarchical facts can be represented in the logic language using lists within lists. Queries can articulate the structure of hierarchical facts or match variables to entire lists.

Compound queries are supported, where multiple subexpressions must be satisfied simultaneously. If a variable appears more than once in a query, it must take the same value in each context.

The logic language can efficiently express relationships among facts. It supports operations like appending lists together using rules defined as facts. The query interpreter can compute the results of appending lists and find all possible combinations of lists that can form a specific result.

Overall, the logic language provides a declarative and efficient way to represent and query facts, supporting complex relationships and inference in a database.

The query interpreter in the logic language performs inference by using the unification operation. Unification is a method of matching a query to a fact, considering variables in both the query and the fact. The interpreter applies unification to match the query with conclusions of facts and hypotheses of other facts in the database. It searches through the space of related facts and returns a successful result if it finds an assignment of values to variables that supports the query.

Pattern matching is used to match queries containing variables with facts that don't have variables. It involves substituting values into the pattern to obtain the expression and checking if it matches the given query.

Queries and facts in the logic language are represented as Scheme lists. An environment, which binds symbols to values, is represented using the Frame class. The unify function performs pattern matching and unification between two expressions, updating the environment with variable-value bindings.

Unification is a recursive process that attempts to find a mapping between two expressions containing variables. It continues unifying corresponding parts of the expressions until a contradiction is reached or all variables are successfully bound.

The logic language can be seen as a prover of assertions in a formal system. Facts establish axioms, and queries must be established by the query interpreter from these axioms. The query interpreter verifies the truth of a query by finding assignments to variables that satisfy all sub-expressions based on the facts.

Search is an essential part of the query interpreter, as it explores the space of possible facts to establish a query. Unification is the primitive operation used for pattern matching during the search process. The search function performs the search procedure for the logic language, recursively applying rules and unifying clauses to find a set of facts that support the query.

During the search, if the first clause is negated, the interpreter checks that unification is not possible. If it is not negated, the interpreter attempts to unify the first clause of a fact with the first clause of the query. If successful, it proceeds to establish the hypotheses of the rule and recursively searches to establish the rest of the clauses in the query.

To avoid confusion between variables with the same name in different facts or queries, variable names are replaced with unique names using the rename_variables function.

The logic example program provides further details, including the user interface and various helper functions.

Next: 4.6 Distributed Computing

Distributed computing involves multiple independent computers coordinating their efforts to perform a joint computation. Communication between computers in distributed systems is done through messages, which are sequences of bytes. These messages adhere to message protocols, which define the rules for encoding and interpreting the messages. The Internet Protocol (IP) is used to transfer messages between machines on the Internet, and the Transmission Control Protocol (TCP) provides reliable and ordered transmission of data over IP.

The client/server architecture is a common model in distributed systems, where a server provides a service and multiple clients communicate with the server to consume that service. Clients issue requests to the server, and the server responds to these requests. The World Wide Web is an example of a client/server architecture, where web browsers (clients) request web pages from web servers and receive responses.

HTTP (Hypertext Transfer Protocol) is a protocol used for communication between web browsers and web servers in the client/server model. HTTP requests, such as GET requests, are used to retrieve specific web pages, and the server responds with an HTTP response, including status codes indicating the success or failure of the request.

In peer-to-peer systems, the division of labor is more equal among all the components of the system. Each computer in the system contributes processing power and memory, and they communicate with each other to perform distributed computations. Peer-to-peer systems require an organized network structure to ensure reliable communication between peers. They are commonly used for data transfer and storage, where each computer contributes to sending data over the network or stores a portion of the data.

Skype is an example of a peer-to-peer application for data transfer, where communications between users are transmitted through a peer-to-peer network composed of other computers running the Skype application. Supernodes in the network help maintain the network structure and handle user logins and logouts.

Distributed systems are used to process large datasets that are stored across multiple machines. In such systems, communication is required to retrieve the data needed for processing. MapReduce is a programming framework used to coordinate distributed data processing. It involves two main components:

1. Map Function: This function is applied to each input and generates intermediate key-value pairs.

2. Reduce Function: This function combines the values for each key and produces the final output.

MapReduce divides the processing tasks among multiple machines, with parallel execution of map and reduce functions. The framework handles communication and coordination between machines, addressing issues like failures and network problems.

In a local implementation of MapReduce using Unix tools, a Python program can be converted into a Unix program by specifying the Python interpreter. The input and output of the program are handled through standard input and standard output, respectively. Unix commands like "chmod" and "pipe" are used to execute and connect programs.

To implement a basic MapReduce framework, Unix programs can be created for the map and reduce tasks, and the whole MapReduce application can be executed by chaining these programs together using pipes.

In a distributed implementation using Hadoop, the open-source MapReduce implementation, computation is distributed across a cluster of machines for efficient parallel processing. Hadoop provides speed, fault tolerance, and monitoring capabilities. Hadoop's streaming interface allows the use of arbitrary Unix programs as map and reduce functions.

To run a MapReduce application with Hadoop, you need to install Hadoop, define the map and reduce functions in Python, copy input data to the Hadoop distributed file system, and execute the application using the Hadoop streaming interface.

For more detailed information and instructions on using Hadoop, you can refer to the Hadoop Streaming Documentation.

Parallel computing is the use of multiple processors or cores to perform computations simultaneously. In the past, processor speeds increased exponentially by increasing the clock frequency, but this approach became limited due to power and thermal constraints. To overcome this limitation, CPU manufacturers started incorporating multiple cores in a single processor, allowing for parallel execution of tasks.

Parallelism has been used in large-scale machines for scientific computing and data analysis. Even personal computers with a single processor core can achieve concurrency through operating systems and interpreters that switch between different tasks. However, with the increasing number of processor cores, individual applications need to take advantage of parallelism to improve performance.

Functional programming, which avoids shared mutable state, lends itself well to parallelism. Pure functions with referential transparency can be evaluated in parallel since their expressions are independent of each other. The MapReduce framework is an example of a parallel programming model that facilitates parallel execution of functional programs.

However, not all problems can be efficiently solved using functional programming. The Berkeley View project identified thirteen common computational patterns, including MapReduce, that require shared state. Parallel programming with shared mutable state introduces challenges and can lead to bugs.

Python provides two means of parallel execution for parallel computing: threading and multiprocessing. Threading allows multiple threads of execution within a single interpreter, while multiprocessing involves spawning multiple independent interpreters or processes. Threads share data but are subject to interpreter limitations, while processes can truly run concurrently if the CPU has multiple cores.

The threading module in Python enables the creation and synchronization of threads. The multiprocessing module provides classes for creating and synchronizing processes. Both modules have similar constructs, allowing threads or processes to execute target functions concurrently.

Shared mutable state in parallel programs can lead to race conditions, where one thread's mutation conflicts with another thread's access to the shared data. To avoid race conditions, shared data that can be mutated and accessed by multiple threads must be protected against concurrent access. Synchronization mechanisms, such as locks and barriers, are used to ensure exclusive access to shared data.

Locks are mechanisms to provide mutual exclusion, allowing only one thread to acquire a lock at a time. The threading module provides a Lock class for synchronization. Threads must acquire the lock before accessing shared data and release it when done.

Barriers divide a program into phases, where threads must wait at the barrier until all threads have reached it before proceeding. Barriers are used to synchronize access to shared data in different phases of the program. The threading module provides a Barrier class with a wait method for this purpose.

Python also offers synchronized data structures, such as the Queue class in the queue module. Synchronized data structures provide thread-safe operations, ensuring that data access is coordinated and consistent across threads.

In summary, parallel computing is essential for achieving better performance in modern processors with multiple cores. Python supports parallel execution through threading and multiprocessing. Synchronization mechanisms like locks, barriers, and synchronized data structures are used to coordinate access to shared mutable state and prevent race conditions.