notebook

May 8, 2023

1 Concepts of Programming Languages

1.1 General Concepts:

1.1.1 Variables and scoping

In programming languages, variables are used to store and manipulate data. They act as containers for values that can be changed during program execution. Variables have a name (identifier) and a data type, which defines the kind of values the variable can store.

Scoping refers to the visibility and lifetime of a variable within a program. There are two main types of scoping:

- Local Scope: Variables declared within a function or a block of code have a local scope. They are only accessible within that function or block and are destroyed once the function or block is exited.
- Global Scope: Variables declared outside of any function or block have a global scope. They can be accessed from anywhere within the program and their lifetime lasts as long as the program is running.

Some languages also have other scoping rules, such as lexical (or static) scoping and dynamic scoping, but local and global scoping are the most common.

[ ]:
```
# Global variable (accessible throughout the entire program)
global_variable = "I am a global variable"

def my_function():
    # Local variable (only accessible within the function)
    local_variable = "I am a local variable"
    print(global_variable)  # This will work, as global_variable is accessible␣
  ↪within the function
    print(local_variable)   # This will also work, as local_variable is␣
  ↪accessible within the function

my_function()

print(global_variable)  # This will work, as global_variable is accessible␣
  ↪outside the function
```

```
print(local_variable)   # This will raise an error, as local_variable is not␣
↪accessible outside the function
```

Lexical (Static) Scoping Lexical scoping, also known as static scoping, is a scoping rule where a variable's scope is determined by its position within the source code. In other words, the scope of a variable is defined by its surrounding block or function at compile time. Lexical scoping is the most common scoping rule used in programming languages, including Python, JavaScript, and C++.

[ ]:
```
x = 10  # Global variable

def outer():
    y = 20  # Enclosing variable (local to outer())

    def inner():
        z = 30  # Local variable
        print(x, y, z)  # Access all variables in the lexical scope

    inner()

outer()
# Output: 10 20 30
```

In this example, the inner() function has access to its local variable z, the enclosing variable y, and the global variable x.

1.1.2 Dynamic Scoping

Dynamic scoping is a scoping rule where a variable's scope is determined by the function call stack during program execution. Under dynamic scoping, a variable is visible within a function if it is defined in that function or any of its calling functions. Dynamic scoping is less common and is not used in Python. However, it is used in some languages like Emacs Lisp and older versions of Perl.

[ ]:
```
# Emacs Lisp

(defvar x 10)  ;; Global variable

(defun outer ()
  (setq y 20)  ;; Dynamic variable (local to outer)
  (inner))

(defun inner ()
  (setq z 30)  ;; Dynamic variable (local to inner)
  (message "x: %d, y: %d, z: %d" x y z))

(outer)
;; Output: "x: 10, y: 20, z: 30"
```

In this example, the inner function has access to its local variable z, the dynamically scoped variable y, and the global variable x.

Python does not support dynamic scoping, so the examples provided use different languages to demonstrate the concept.

Primitive data types Primitive data types are the basic building blocks of data in a programming language. They are the simplest data types that can be used to create more complex data structures. The exact set of primitive data types varies between programming languages, but some common ones include:

- Integer (int): Represents whole numbers, both positive and negative. Examples: 42, -3, 0
- Floating-point (float, double): Represents real numbers, i.e., numbers with a decimal point. Float usually has less precision (less number of decimal places) than double. Examples: 3.14, -0.003, 7.0
- Boolean (bool): Represents a binary value, either true or false.
- Character (char): Represents a single character, such as a letter, digit, or special symbol. Examples: 'A', '7', '%'

Some languages may also include other primitive data types, such as byte (for small integers), short (for medium-sized integers), and long (for large integers), among others.

Understanding variables, scoping, and primitive data types is crucial for programming, as they form the foundation for working with data in any programming language.

[ ]:
```python
# Integer
my_integer = 42
print(my_integer)  # Output: 42

# Floating-point
my_float = 3.14
print(my_float)  # Output: 3.14

# Boolean
my_boolean = True
print(my_boolean)  # Output: True

# Character (In Python, characters are represented as strings of length 1)
my_char = 'A'
print(my_char)  # Output: A
```

1.1.3 Type Systems

A type system is a set of rules that a programming language follows to manage data types and their interactions. Type systems are important because they help catch potential errors during the development process and ensure that operations on data make sense. There are two primary aspects of type systems we will discuss: static/dynamic typing and type inference.

Background Knowledge - "Declared or inferred" "Declared or inferred" refers to how the type of a variable is determined in a programming language. It can either be explicitly declared by the programmer or implicitly inferred by the language's type system. Let's break down both terms:

- Declared: When the type of a variable is declared, it means the programmer explicitly specifies the type of the variable when it is defined. This is common in statically-typed languages like Java, C++, and C#, where the type of a variable must be known at compile-time. Declaring a variable's type helps the compiler understand what kind of data the variable will store and catch potential type-related errors early.

[ ]:
```
# Java

int x = 10; // The type 'int' is explicitly declared by the programmer
```

- Inferred: Type inference is when the programming language automatically deduces the type of a variable based on the value it is assigned or the context in which it is used. Some statically-typed languages, like Haskell, Scala, and modern versions of C++ and C#, have type inference mechanisms that allow the programmer to omit explicit type declarations in certain situations. This can make the code more concise and easier to read and maintain.

[ ]:
```
# C++

auto x = 10; // The type 'int' is inferred by the compiler based on the␣
 ↪assigned value
```

Static/dynamic typing Static typing and dynamic typing refer to how and when a programming language checks the types of variables and expressions.

Static typing: In statically-typed languages, type checking occurs at compile-time. Variables have a fixed type, which must be declared explicitly or implicitly when a variable is defined. Examples of statically-typed languages include Java, C++, and C#.

Pros:

- Early error detection: Since type checking is performed at compile-time, many type-related errors can be caught before the program runs. This can save time and effort in debugging and fixing issues during runtime.
- Performance: Statically-typed languages usually have better performance because the types are known at compile-time, allowing the compiler to optimize the generated code. This can result in faster execution and less memory usage.
- Readability and maintainability: In statically-typed languages, variable types are explicitly declared or inferred at the time of definition, which can make the code more self-explanatory and easier to understand. This can be particularly helpful in large codebases or when working with a team.
- Better tooling: With static types, Integrated Development Environments (IDEs) and other development tools can provide better code navigation, autocompletion, and refactoring capabilities.

Cons:

- Verbosity: Statically-typed languages often require more code to declare and manage types, which can result in longer and more complex code.
- Less flexibility: Since variables have a fixed type, it can be more difficult to write generic code or change the data types used in a program without modifying the code.

Dynamic typing In dynamically-typed languages, type checking occurs at runtime. Variables can change their type during the execution of the program, and there is no need to declare the type when defining a variable. Examples of dynamically-typed languages include Python, Ruby, and JavaScript.

Pros:

- Ease of use: Dynamically-typed languages are often easier to learn and use because they do not require explicit type declarations. This can lead to shorter and more concise code.
- Flexibility: With dynamic typing, variables can change their type during program execution, making it easier to write generic code or modify data types without changing the code. This can be particularly helpful in rapid prototyping and exploratory programming.
- Runtime dynamism: Dynamically-typed languages allow for greater runtime flexibility, enabling features like dynamic code loading, runtime code modification, and reflection.

Cons:

- Late error detection: Type-related errors might only be discovered during runtime, which can make debugging and fixing issues more time-consuming and challenging.
- Performance: Dynamically-typed languages can have worse performance compared to statically-typed languages due to the overhead of runtime type checking and the inability to optimize the code at compile-time.
- Readability and maintainability: In dynamically-typed languages, it can be harder to understand the types of variables and their expected behavior just by looking at the code. This can make the code more challenging to maintain, especially in large projects or when working with a team.
- Limited tooling: Development tools for dynamically-typed languages might have limited code navigation, autocompletion, and refactoring capabilities compared to those for statically-typed languages due to the absence of type information at compile-time.

Type inference Type inference refers to the ability of a programming language to automatically deduce the type of a variable or an expression based on the context. This is often used in statically-typed languages to allow the programmer to omit explicit type annotations, making the code shorter and more readable.

Languages with strong type inference capabilities include Haskell, Scala, and Kotlin. Some languages, like Java and C#, have limited type inference capabilities using the var keyword (Java 10+ and C# 3.0+).

[ ]:
```
// In Java, you must declare the type of a variable explicitly
int myInteger = 42;
String myString = "Hello, world!";

// This will result in a compile-time error due to incompatible types
myString = myInteger;
```

[ ]:
```
# In Python, you don't need to declare the type of a variable explicitly
my_variable = 42
print(my_variable)  # Output: 42

# The type of the variable can change during the execution of the program
my_variable = "Hello, world!"
print(my_variable)  # Output: Hello, world!
```

[ ]:
```
// In Kotlin, you can use 'val' or 'var' for type inference
val myInteger = 42  // The compiler infers the type as 'Int'
val myString = "Hello, world!"  // The compiler infers the type as 'String'

// This will result in a compile-time error due to incompatible types
// myString = myInteger
```

1.2 Object-oriented concepts

1.2.1 Classes and objects

In object-oriented programming (OOP), a class is a blueprint for creating objects, which are instances of that class. Classes define the properties (attributes) and behaviors (methods) that the objects will have. Objects are instances of classes, and each object has its own state (values of its attributes) and can perform actions using its methods.

[ ]:
```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        print(f"{self.name} says: Woof!")

# Create an object (instance of the Dog class)
my_dog = Dog("Buddy", "Golden Retriever")

# Access object's attributes and methods
print(my_dog.name)   # Output: Buddy
print(my_dog.breed)  # Output: Golden Retriever
my_dog.bark()        # Output: Buddy says: Woof!
```

1.2.2 Inheritance

Inheritance is a mechanism in OOP that allows one class to inherit properties (attributes) and behaviors (methods) from another class. This helps in reusing code and creating a hierarchy of related classes.

- Single inheritance: A class inherits from only one base class (parent class).

[ ]:
```
# Single inheritance
class Animal:
    def __init__(self, name):
        self.name = name

class Dog(Animal):
    def bark(self):
        print(f"{self.name} says: Woof!")

# Create an object (instance of the Dog class)
my_dog = Dog("Buddy")
my_dog.bark()  # Output: Buddy says: Woof!
```

- Multiple inheritance: A class inherits from multiple base classes.

[ ]:
```
class Parent1:
    def greet1(self):
        print("Hello from Parent1 class")

class Parent2:
    def greet2(self):
        print("Hello from Parent2 class")

class Child(Parent1, Parent2):
    pass

c = Child()
c.greet1()
c.greet2()
# Output:
# Hello from Parent1 class
# Hello from Parent2 class
```

- Interface inheritance: A class inherits only the signatures (method declarations) of the methods from an interface, but not their implementation.

[ ]:
```
from abc import ABC, abstractmethod

class GreetInterface(ABC):
    @abstractmethod
    def greet(self):
        pass

class GreetImplementation(GreetInterface):
    def greet(self):
        print("Hello from GreetImplementation class")

g = GreetImplementation()
g.greet()
# Output:
# Hello from GreetImplementation class
```

- Abstract classes: Classes that cannot be instantiated and are meant to be subclassed. They can have both implemented and unimplemented methods.

[ ]:
```
from abc import ABC, abstractmethod

class AbstractGreet(ABC):
    @abstractmethod
    def greet(self):
        pass

    def hello(self):
        print("Hello from non-abstract method")

class GreetImplementation(AbstractGreet):
    def greet(self):
        print("Hello from GreetImplementation class")

g = GreetImplementation()
g.greet()
g.hello()
# Output:
# Hello from GreetImplementation class
# Hello from non-abstract method
```

- Prototype-based inheritance: In some languages (like JavaScript), objects can inherit directly from other objects without the need for classes.

[ ]:
```
// Python does not support prototype-based inheritance natively, as it is a␣
 ↪class-based language.
// However, JavaScript is an example of a language that uses prototype-based␣
 ↪inheritance:

const parent = {
  greet: function() {
    console.log("Hello from Parent object");
  }
};
```

```
const child = Object.create(parent);
child.greet();
// Output:
// Hello from Parent object
```

- Class-based inheritance: Inheritance based on classes, used in languages like Java, C++, and Python.

[ ]:
```
class Parent:
    def greet(self):
        print("Hello from Parent class")

class Child(Parent):
    def greet(self):
        super().greet()
        print("Hello from Child class")

c = Child()
c.greet()
# Output:
# Hello from Parent class
# Hello from Child class
```

1.2.3 Polymorphism

Polymorphism is the ability of different objects to respond to the same method call in a way that is specific to their individual types. It allows for more abstract and flexible code by enabling the same interface to be used with different types.

- Dynamic dispatch: The process of selecting the appropriate method implementation at runtime based on the type of the object.
- Late binding: The process of resolving method calls to their actual implementation at runtime, as opposed to compile time.

[ ]:
```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        print("Woof!")

class Cat(Animal):
    def speak(self):
        print("Meow!")

def make_animal_speak(animal):
    animal.speak()

# Polymorphism in action
dog = Dog()
cat = Cat()

make_animal_speak(dog)  # Output: Woof!
make_animal_speak(cat)  # Output: Meow!
```

These examples demonstrate the core concepts of object-oriented programming, including classes, objects, inheritance, and polymorphism. Keep in mind that the specific syntax and behavior may vary between different languages, so always consult the language documentation for details.

1.3 Functional concepts:

1.3.1 Pure functions and referential transparency

A pure function is a function that meets the following criteria:

- The output depends only on its input arguments, without any hidden state or external dependencies. It does not cause any side effects, such as modifying global variables or changing the input arguments.
- Referential transparency is a property of a function where it can be replaced with its output value without affecting the behavior of the program. In other words, given the same input, a referentially transparent function will always produce the same output.

[ ]:
```
# A pure function with referential transparency
def add(x, y):
    return x + y

result1 = add(2, 3)  # Output: 5
result2 = add(2, 3)  # Output: 5 (same output for the same input)
```

1.3.2 Anonymous functions and lambda expressions

An anonymous function is a function that is defined without a name. They are usually short and simple functions that can be used as arguments to other functions or as a part of a more complex expression. Lambda expressions are a way to create anonymous functions in some programming languages, including Python.

[ ]:
```
# A lambda function that adds two numbers
add = lambda x, y: x + y

result = add(2, 3)  # Output: 5
```

1.3.3 Higher-order functions

Higher-order functions are functions that can take other functions as arguments or return them as results. They are a fundamental concept in functional programming and enable powerful techniques such as composition and partial application.

Some common higher-order functions include map, filter, and reduce.

- map: Applies a function to each item in an input list and returns a new list with the results.
- filter: Filters the items in an input list based on a predicate function and returns a new list with the items that pass the predicate.
- reduce: Applies a function to the items in an input list in a cumulative way, reducing the list to a single value.

[ ]:
```
numbers = [1, 2, 3, 4, 5]

# Using map to square each number in the list
squared_numbers = list(map(lambda x: x * x, numbers))
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]

# Using filter to get only the even numbers in the list
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4]

# Using reduce to find the product of all numbers in the list
from functools import reduce
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 120
```

These examples demonstrate the core concepts of functional programming, including pure functions, referential transparency, anonymous functions, lambda expressions, and higher-order functions. Keep in mind that the specific syntax and behavior may vary between different languages, so always consult the language documentation for details.

1.4 Advanced Topics

1.4.1 Memory Management

Memory management is the process of handling the allocation and deallocation of memory during the execution of a program. Memory management can be performed manually by the programmer, or automatically by the language runtime through mechanisms like garbage collection.

Heap and stack The heap and stack are two areas of memory used by programs:

- Heap: The heap is an area of memory used for dynamic memory allocation. Objects that are created during the runtime of a program are typically stored on the heap. The size of the heap can grow or shrink during the execution of a program.
- Stack: The stack is an area of memory used for static memory allocation. It stores local variables and function call information. The stack follows a Last-In-First-Out (LIFO) structure, and its size is determined at compile time.

Manual cleanup vs. garbage collection Manual cleanup: In languages like C and C++, the programmer is responsible for allocating and deallocating memory. This allows for more control over memory usage but can lead to issues like memory leaks, dangling pointers, and double-free errors.

[ ]:
```
// Example in C
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *array = malloc(10 * sizeof(int));  // Allocate memory on the heap

    // Use the array...

    free(array);  // Deallocate memory when it's no longer needed
    return 0;
}
```

Garbage collection: In languages like Java, Python, and C#, memory management is mostly automated through garbage collection. The garbage collector automatically identifies and deallocates objects that are no longer in use, which reduces the chances of memory leaks and related issues.

[ ]:
```
// Example in Java
public class Main {
    public static void main(String[] args) {
        int[] array = new int[10];  // Allocate memory on the heap

        // Use the array...

        // No need to manually deallocate memory; the garbage collector will␣
 ↪handle it
    }
}
```

1.4.2 Reflection

Reflection is a mechanism that allows programs to inspect and interact with their own structure and behavior at runtime. This includes examining the types, methods, and fields of objects, as well as creating new instances, invoking methods, and modifying fields.

[ ]:
```
class MyClass:
    def __init__(self, x):
        self.x = x

    def print_x(self):
        print(self.x)

# Create an instance of MyClass
obj = MyClass(42)

# Get the type of the object
obj_type = type(obj)
print(obj_type)  # Output: <class '__main__.MyClass'>
```

```
# Get the list of methods and attributes of the object
print(dir(obj))

# Invoke the print_x method using reflection
getattr(obj, 'print_x')()  # Output: 42
```

1.4.3 Generics

Generics are a mechanism that allows the creation of classes, interfaces, and functions with placeholders for the types they operate on. This enables code reuse, type safety, and better separation of concerns. Generics are supported in languages like Java, C#, and Kotlin.

[ ]:
```
// A simple generic class in Java
public class Box<T> {
    private T contents;

    public Box(T contents) {
        this.contents = contents;
    }

    public T getContents() {
        return contents;
    }

    public void setContents(T contents) {
        this.contents = contents;
    }
}

public class Main {
    public static void main(String[] args) {
        Box<Integer> intBox = new Box<>(42);
        System.out.println(intBox.getContents());  // Output: 42

        Box<String
```

1.4.4 Language Implementation (Compilers/Interpreters)

A compiler is a program that translates source code written in a high-level programming language into a lower-level language, usually machine code or assembly language. An interpreter, on the other hand, is a program that directly executes the instructions in the source code without translating it into a lower-level language.

The process of language implementation typically consists of the following stages:

Lexical analysis Lexical analysis is the process of converting the input source code into a sequence of tokens. A token is a representation of a basic syntactic element in the source code, such as an identifier, keyword, or operator.

```
[ ]: # Example input (Python source code)
     x = 42

     # Example output (tokens)
     [('identifier', 'x'), ('operator', '='), ('number', 42)]
```

Parsing Parsing is the process of analyzing the sequence of tokens generated by the lexical analyzer to produce a parse tree, which represents the syntactic structure of the source code. This step usually involves checking the code for syntax errors.

```
[ ]: # Example input (tokens)
     [('identifier', 'x'), ('operator', '='), ('number', 42)]

     # Example output (parse tree)
     Assignment(Variable('x'), Number(42))
```

Semantic analysis Semantic analysis is the process of checking the code for semantic errors, such as type mismatches and undeclared variables. This step also involves activities like type checking, static analysis, and symbol table construction.

```
[ ]: # Example input (parse tree)
     Assignment(Variable('x'), Number(42))

     # Example output (annotated parse tree)
     Assignment(Variable('x', type='int'), Number(42, type='int'))
```

1.4.5 Code optimization

Code optimization is an optional step in the compilation process that aims to improve the performance of the generated code. This can include techniques such as constant folding, dead code elimination, and loop unrolling.

```
[ ]: # Example input (annotated parse tree)
     Assignment(Variable('x', type='int'), Add(Number(20, type='int'), Number(22,␣
      ↪type='int')))

     # Example output (optimized parse tree)
     Assignment(Variable('x', type='int'), Number(42, type='int'))
```

Code generation Code generation is the final step in the compilation process, where the compiler generates the target code (usually machine code or assembly language) from the optimized parse tree or intermediate representation.

```
[ ]: # Example input (optimized parse tree)
     Assignment(Variable('x', type='int'), Number(42, type='int'))
```

```
# Example output (assembly code)
MOV eax, 42
MOV [x], eax
```

These examples demonstrate a simplified view of the different stages involved in the process of implementing programming languages using compilers and interpreters. In practice, these stages can be more complex and may involve additional steps, such as intermediate code generation and peephole optimization. The specific details and techniques used in each stage may vary depending on the programming language and the target platform.