# Programming, Data Structures, and Algorithms in Python: Week 2

This document provides a detailed summary of the concepts covered in the second week, focusing on Python's basic data types, control flow mechanisms, and the structure and behavior of functions.

## 1. Basic Python Syntax and Program Structure

A Python program is a sequence of statements executed by the Python interpreter from top to bottom.

### Assignment Statements

The most fundamental statement is assigning a value to a name (or variable).

* **Syntax:** name = expression
* **Example:** x = 5, y = 2 \* x
* **Execution:** The expression on the right-hand side is evaluated first, and the resulting value is assigned to the name on the left.
* **Updating Values:** A name can appear on both sides, which signifies an update. In j = j + 5, the old value of j is used to compute the new value, which is then reassigned to j.

### Program Organization

While Python allows mixing function definitions and statements, the recommended structure for readability and to avoid errors is:

1. **Function Definitions:** Place all def blocks at the top of the file.
2. **Main Statements:** Follow with the main part of your code that executes the program's logic.

A function must be **defined** before it is **called**. By placing all definitions first, the interpreter processes them and makes them available for any subsequent statement to use.

## 2. Fundamental Data Types

Every value in Python has a **type**, which determines the operations that can be performed on it. Unlike other languages, names in Python do not have a fixed type; they inherit the type of the value currently assigned to them.

### Numeric Types: int and float

Python distinguishes between two main types of numbers:

* **int (Integers):** Whole numbers, both positive and negative (e.g., 5, -10, 0).
* **float (Floating-Point Numbers):** Numbers with a fractional part (e.g., 3.14, -0.5, 10.0).

This distinction exists because they are represented differently in computer memory.

* **Arithmetic Operations:** + (addition), - (subtraction), \* (multiplication), / (division).
  + Division (/) always produces a float.
  + Python allows mixing int and float in operations; the result will be a float.
* **Integer-Specific Operations:**
  + // (Integer Division): Gives the quotient. E.g., $9 // 5$ results in 1.
  + % (Modulo): Gives the remainder. E.g., $9 \% 5$ results in 4.
* **Exponentiation:** \*\* raises a number to a power. E.g., $3 \*\* 4$ is $3^4 = 81$.
* **Math Library:** Advanced functions like log, sqrt, and sin are available by importing the math module: from math import \*.

### Logical Type: bool

* Represents truth values.
* **Values:** True and False (case-sensitive).
* **Logical Operations:** not, and, or.
* **Comparison Operators:** These operators evaluate to a bool.
  + == (Equal to)
  + != (Not equal to)
  + < (Less than), > (Greater than)
  + <= (Less than or equal to), >= (Greater than or equal to)

## 3. Sequence Types: str and list

Sequences are ordered collections of items.

### Strings (str)

Strings are sequences of characters used to represent text.

* **Literals:** Defined using single ('...'), double ("..."), or triple ('''...''' or """...""") quotes. Triple quotes are useful for multi-line strings or strings containing both single and double quotes.
* **Indexing:** Access individual characters using their position, starting from 0. Negative indices count from the end (-1 is the last character). E.g., if s = 'hello', then s[0] is 'h' and s[-1] is 'o'.
* **Slicing:** Extract a substring using [start:end]. The slice includes the character at start and goes up to, but does not include, the character at end.
  + s[1:4] gives 'ell'.
  + s[:3] (omitting start) gives 'hel'.
  + s[2:] (omitting end) gives 'llo'.
* **Concatenation:** The + operator joins strings together. E.g., 'hello' + ' ' + 'world' results in 'hello world'.
* **Length:** len(s) returns the number of characters in the string s.

### Lists (list)

A list is a versatile sequence that can hold values of any type, including other lists.

* **Literals:** Defined using square brackets [], with items separated by commas. E.g., [1, 2, 5, 10], ['Anand', 'Charles'].
* **Heterogeneous:** Lists can contain mixed types: [3, True, 'hello'].
* **Indexing and Slicing:** Works exactly like strings.
* **Nested Lists:** A list can contain other lists, creating multi-dimensional structures. You can access elements by chaining indices, e.g., nested[0][1].

### Mutability vs. Immutability

This is a crucial concept in Python.

* **Immutable:** The value cannot be changed in place. If you want to modify it, you must create a new value.
  + **Types:** int, float, bool, str.
  + **Example:** You cannot change a character in a string with s[3] = 'p'. You must build a new string using slices and concatenation: s = s[:3] + 'p!'.
* **Mutable:** The value *can* be changed in place without creating a new object.
  + **Types:** list.
  + **Example:** You can directly modify an element: my\_list[2] = 4. This changes the list itself.

### Assignment and Object References

The distinction between mutable and immutable types deeply affects how assignment works.

* **Immutable Assignment:** y = x creates a **copy** of the value. Subsequent changes to x will not affect y.
* **Mutable Assignment:** list2 = list1 does **not** create a copy. Instead, both list1 and list2 now **refer to the same list object** in memory. Modifying the list through one name will affect the other.

How to Copy a List:

To create a true, independent copy of a list, use a full slice:

list2 = list1[:]

**Checking Equality:**

* ==: Checks if the **values** of two objects are the same.
* is: Checks if two names refer to the **exact same object** in memory.

## 4. Control Flow

Control flow statements alter the default top-to-bottom execution of a program.

### Conditional Execution: if, elif, else

* if condition:: Executes the indented block of code only if condition is True.
* else:: If the if condition is False, the else block is executed.
* elif condition:: Stands for "else if". It allows for checking multiple conditions in sequence. This avoids deep nesting of if-else statements.

Truthiness:

Non-Boolean values can be interpreted as True or False in a condition:

* **False:** The number 0, the empty string "", the empty list [].
* **True:** Any non-zero number, any non-empty string or list.

### Loops: for and while

* **for loop:** Repeats a block of code for a **fixed number of times**, iterating over a sequence.
  + for i in my\_list: iterates through each element of my\_list.
  + for i in range(n): iterates n times, with i taking values from 0 to n-1.
* **while loop:** Repeats a block of code **as long as a condition is true**.
  + Used when the number of repetitions is not known in advance.
  + It's crucial to ensure that the code inside the loop makes progress toward making the condition False, otherwise it will result in an infinite loop.

While any for loop can be simulated with a while loop, using for is more readable and idiomatic when the number of iterations is known.

## 5. Functions

A function is a named, reusable block of code that performs a specific task.

* **Definition:** def function\_name(parameter1, parameter2):
* **Arguments:** When a function is called, the values passed to it (arguments) are assigned to its parameters. This assignment follows the same rules for mutable and immutable types.
  + Changes made inside the function to **immutable** arguments (like numbers or strings) do not affect the original variables outside.
  + Changes made inside the function to **mutable** arguments (like lists) **will** affect the original list. This is often called a "side effect".
* **return Statement:** Ends the function's execution and sends a value back to the caller. A function can have multiple return statements. If a function reaches its end without a return statement, it implicitly returns None.
* **Scope:** Names defined inside a function (local scope) are separate from names defined outside (global scope). A name used inside a function does not conflict with a name used outside.
* **Recursion:** A function is recursive if it calls itself. This is a powerful technique for solving problems that can be broken down into smaller, self-similar sub-problems, like calculating factorials.

### Example: Finding Prime Numbers

These concepts can be combined to solve problems. For instance, to find prime numbers:

1. **factors(n):** A function to find all factors of n.  
   def factors(n):  
    flist = []  
    for i in range(1, n + 1):  
    if n % i == 0:  
    flist.append(i) # Use append to add to list  
    return flist
2. **isprime(n):** A function that uses factors() to check for primality.  
   def isprime(n):  
    return factors(n) == [1, n]
3. **primesupto(n):** A function using a for loop to find all primes up to a limit n.  
   def primesupto(n):  
    plist = []  
    for i in range(1, n + 1):  
    if isprime(i):  
    plist.append(i)  
    return plist
4. **nprimes(n):** A function using a while loop to find the first n prime numbers.  
   def nprimes(n):  
    count = 0  
    i = 1  
    plist = []  
    while count < n:  
    if isprime(i):  
    count = count + 1  
    plist.append(i)  
    i = i + 1  
    return plist

This structured approach, breaking a problem into smaller functions, is key to writing clear, correct, and maintainable code.