Advanced Programming

Python

Ramaseshan Ramachandran

1. Program Structure
2. Naming Conventions
3. Python

String

List

Dictionary

generators

COMMON STRUCTURES

- Preprocessor directives (optional): Instructions for the compiler or interpreter, often used for header files or conditional compilation
- Declarations: Definitions of variables, functions, and other elements, specifying their names and data types.
- Functions:Reusable blocks of code that perform specific tasks, often with parameters and a return value.
- Main function:The entry point of the program, where execution begins.
- Statements: Instructions that perform actions, such as assignments, calculations, input/output, and control flow.
- Control structures: Statements that control the flow of execution, such as conditional statements (if/else) and loops (for/while)
- Comments: Non-executable text explaining code functionality

SAMPLE CODE IN PYTHON

```python
def main():
    # Declarations
    x = 10  #not the best way to name a variable
    y = 5

    # Statements
    sum = x + y
    print("The sum is:", sum)

# Call the main function to start execution
# Entry point
if __name__ == "__main__":
    main()
```

SAMPLE CODE IN C

```c
#include <stdio.h>

// Entry point
int main() {
    // Declarations
    int x = 10;
    int y = 5;

    // Statements
    int sum = x + y;
    printf("The sum is: %d\n", sum);

    return 0;
}
```

SAMPLE CODE IN SWIFT

```swift
// Entry point
func main() {
    // Declarations
    var x = 10
    var y = 5

    // Statements
    let sum = x + y
    print("The sum is: \(sum)")
}

main()
```

SAMPLE CODE IN RUST

```rust
// Entry point
fn main() {
    // Declarations
    let x = 10;
    let y = 5;

    // Statements
    let sum = x + y;
    println!("The sum is: {}", sum);
}
```

NAMING CONVENTIONS

Set of rules for choosing names for variables, functions, classes, and other elements in code

Naming Conventions	Explanation	Examples
Camel Case	First word in lowercase, subsequent words start with uppercase	firstName, lastName, middleName, totalBalance
Pascal Case	All words start with uppercase. Often used for class names, enums, and sometimes functions.	FirstName, LastName, MiddleName, TotalBalance
Snake Case	Words separated by underscores. Common in Python, C++, and other languages	first_name, last_name, middle_name, total_balance
Kebab Case	Words separated by hyphens. Used in file paths, URLs, and sometimes code elements	first-name, last-name, middle-name, total-balance
Hungarian Notation	Prefixes encode data type. Less common in modern practice due to potential for confusion	strName, intAge, btnSubmit

NAMING CONVENTIONS - OFFICIAL GUIDES

- **Python**: Official Python Style Guide

PROGRAM STRUCTURE IN PYTHON

Common program structure found in many languages

Entry point	Starting point of the program's execution
Declarations	Statements that define variables, constants, functions, or other program elements
Statements	Instructions that perform actions or calculations
Control Structure	Elements that control the flow of execution, such as conditional statements (if/else) and loops (for/while)
Comments	Non-executable text explaining functionality of the code (for self and co-developer's understanding)

FUNDAMENTAL (PRIMARY) SCALAR OBJECTS

Scalar Objects	Explanation	Examples
Integer type	Signed and unsigned values	$no\_students = 29$
Float	Represents decimal values	$pi = 3.142$
String	Holds sequence of characters	$name = 'Whats\ in\ a\ name'$
Boolean	Logical type (True or False)	$is\_awake = False$
None Type	Absence of a value	$ipl\_tickets = None$

OPERATIONS ON THE SCALAR TYPES

Operations	Operators
Arithmetic operations	$+-/\%$
Equality Checks	You can compare them using $==$ and $!=$ operators
Ordering	They can be ordered using $<,>,<=$, and $>=$ operators (except for None Type)
Type Checking	Use the type() function to find their type
Type Conversion	Convert between different scalar types using functions like int(), float(), and str()

INPUT AND OUTPUT STATEMENTS

```
if __name__ == '__main__':
    first_name = input('Enter First Name: ')
    last_name = input('Enter Last Name: ')
    print(first_name,' ', last_name)

    print('First name is ' + first_name + 'and ' + \
          'Last name is ', last_name )

    # What happens in this code segment?
    # Error free?
    x = input('Enter x value as int: ')
    y = input ('Enter y value as int: ')
    print(x+y)
```

INDENTATION

- Code blocks in Python are indented
- Many languages use braces {} for blocks, but Python relies only on indentation for code blocks
- Enhances code readability
- Recommended spaces for indentation is 4 (four). You may use tab also. Remain consistent in the indentation[1]
- Mixing of spaces and tabs is not recommended
- Statements ending with colon(:) can be considered as the starting point for a code block(s). Code user it should be indented

[1] Many IDEs convert tabs into spaces (the number of spaces can be configured)

INDENTATION EXAMPLE

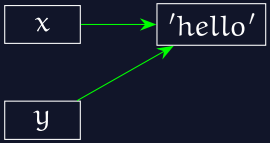
```
def indent_code() -> str:
    # Indented code within the function
    print(’This is inside the function’)

    for i in range(5):
        # Indented code within the loop
        print(i)

    return ’done’
```

STRING

- Python strings are **immutable** - Once stored, the content is fixed
- They can be modified - Create new strings to reflect changes
- A character in a string can be using [ ]
- Zero-based indexing
- -1 refers to the last character in the string

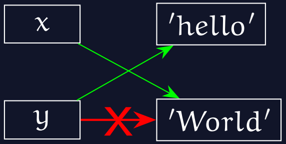
$$x =' hello'$$

$$y = x$$



$$x =' world'$$

$$y = ?$$



STRINGS

```
if __name__ == ’__main__’:
my_string =  ’hello’

my_string = "hello" "world" # concatenation - valid

# Trying to change a character doesn’t work
my_string[0] = "H"  # Error! Strings are immutable

#length of a string
print(len(my_string))

#locate the character(s) using  indexes
print(my_string[1]) #output = ’e’
```

STRING SLICING

```
# slicing  -
# extracting parts of a sequence [start:end:step]
#        +---+---+---+---+----+
#        | H | e | l | l | o |
#        +---+---+---+---+----+
#        0   1   2   3   4
#       -5  -4  -3  -2  -1
if __name__ == '__main__':
my_string =  'hello'
print(my_string[2:4])
# output = 'll' ?   What are the start
# and end indexes used?
print(my_string[:4])    # output = 'hell'
print(my_string[2:])    # output = ?
#what does the following code do?
print(my_string([-1])
```

LIST

- Compound data type[2]
- Group of items belonging to the same type or different types
- Items are separated by a comma and enclosed with in square brackets [ ]
- Usually the items in a list have the same type
- Lists are mutable - while mutability is flexible, be mindful of unintended side effects when modifying lists, as changes can affect other parts of your code that reference the same list

[2] An Informal Introduction to Python

LIST EXAMPLES

```
if __name__ == ’__main__’:
    string_list = [’Hello’, ’World’]
    print(string_list) # output - [’Hello’, ’World’]

    num_list = [1,2,3,4,5]
    #concatenation
    num_list.append(12) # num_list = [1,2,3,4,5,12]
    #Slicing is similar to string slicing
```

LISTS ARE MUTABLE

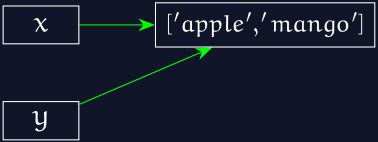
```
>>> #Read-Eval-Print-Loop (REPL)
>>> fruits = ["apple", "banana", "cherry"]
>>> fruits[1] = "mango"  # Change the second element
>>> fruits
["apple", "mango", "cherry"]
>>>
>>> # Remove the first occurrence of "mango"
>>> fruits.remove("mango")
>>> fruits
["apple", "cherry"]
>>> fruits[:] #output - a copy of fruit
>>> fruit[:2] # output = ?
>>> fruit[1:] = #output=?
>>> fruit[1:len(fruits)] # output=?
some_fruits = fruits[1:2]
```

ASSIGNMENT OF LISTS

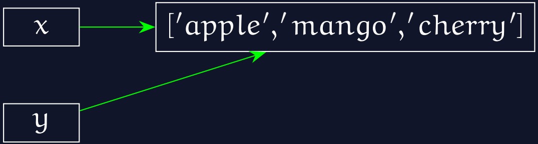
- Python Lists are **mutable** - Once stored, the content is fixed
- Slicing is same as in string
- Zero-based indexing
- An item in a list can be using [ ]
- -1 refers to the last item in the list

$$x = ['apple','mango']$$
$$y = x$$

$$x.append('cherry')$$
$$y = ?$$



Python



ADPG

OPERATIONS ON LISTS

We can insert, delete, modify and copy a slice of the contents of a list

```
>>> fruits = [’pineapple’, ’mango’, ’cherry’]
>>> fruits.insert[0,’berry’]
>>> fruits[3] = ’blueberry’ #cherry is replaced by blueberry
>>> fruits.remove(’mango’) # Remove by occurrence
>>> del fruits[0]  #Remove the element at index
>>> flruits.clear()
```

DICTIONARY

- Another built-in collection object
- Dictionaries are mutable - you can add, remove, or modify key-value pairs after creation
- Keys don't maintain any specific order. Accessing them relies on the key itself, not their position
- Items are stored key-value pairs
- Each key acts as a unique identifier and there cannot be any duplicate key
- Efficient lookup mechanisms to retrieve any item using the key or value
- You can store other dictionaries within a dictionary, creating nested data structures for complex relationships

SAMPLE CODE

```
>>> person = {"name": "Ram", "age": 25, "city": "CHN"}
>>> person["name"] = ’Alice’  # Modify value
>>> print(person["name"])# Output: Alice
>>> person["age"] = 35  # Modify value
>>> #Add new KV pair
>>> person["skills"] = ["coding", "writing"]
>>> person
>>> age = my_dict.get("age")  # Retrieves the value 30
>>> age
>>> # Handling missing keys:
>>> value = my_dict.get("hobby", "Not specified")
>>> value
’Not specified’
```

GENERATORS

Generator Expressions

- Concise way to create iterators: Generate values on demand.
- Syntax: (expression for item in iterable)
- Useful for processing sequences efficiently.

```
if __name__ == ’__main__’:
    numbers = [1, 2, 3, 4]
    number_string = "-".join(str(num) for num in numbers)
    # Output: "1-2-3-4"
```

NOT THE LAST SLIDE