# Quickstart on Programming

Joo-Hyun Paul Kim

June 17, 2023

## Contents

# 0 Preface

This is a short (yet hopefully comprehensive) guide on getting started with programming for beginners (in ANY GENERIC PROGRAMMING LANGUAGE).

This is mostly for beginners, so I won't go into too much detail.

Most code examples shown will be in pseudocode, that is, a language that does not exist, yet should capture the flavor of "generic" programming languages.

# 1 Important Concepts in Programming

Programming is the act of communicating your instructions to a computer. **Computers tend to be precise**, and hence, will do ANYTHING that it is instructed to do, even the mistakes in your instructions.

Think of telling a robot how to make a toast. You might tell it to put the bread on the plate and spread jam over it, and the robot proceeds to take a loaf of bread (not sliced), and pour jam over it straight from the jar.

There are *thousands* of programming languages, but turns out most of them have a similar construct; this means you could learn one programming language, and learn the other ones much more easily.

The following is a list of concepts which most programming languages share:

- Comments
- Variables and Data Type
- List and Tuple
- Conditional
- Loops
- Function

Additionally, quite a lot of languages[1] also have:

- Class and Object
- Lambda Calculus

# 2 Comments

While **comments** are not important for programming itself, it is a valuable tool for communicating with other programmers, or with yourself in the future. Comments are usually explanation that the compiler or interpreters *ignore*. Consider the following code[2]:

```
// fibo(n) returns nth Fibonacci number
fn fibo (n : u32) -> u32 {
    if (n == 1) {
        return 1;
    } else if (n == 2) {
        return 1;
    }
    // Recursion
    fibo (n - 2) + fibo(n - 1)
}
```

The double slash in the fist line explains what this function does. I've also indicated that this function uses "recursion" to compute Fibonacci number.

Comments are also used for temporarily disabling a line of code:

```
// This code prints 1,
// since the previous line is ignored by commenting.
a = 1
//a = 4 + 2
print(a)
```

# 3 Variables and Data Type

## 3.1 Numeric Types

Whatever you are doing, it is likely that you will be dealing with numbers[3]

[1] Python, Ruby, Java, C++, etc.

[2] This is a piece of code in rust.

[3] After all, you are using a *computer*.

### 3.1.1 Integer ($\mathbb{Z}$)

Consider the following pseudocode.

```
// Dynamically typed languages (eg: Python)
a = 1
// Statically typed languages (eg: C)
int a = 1;
```

`a` is a **variable** that contains the value 1.

In most dynamically typed languages (like Python and Javascript), the interpreter and compilers will figure out just from `a = 1` that you are taking the variable `a` to hold an integer.

On the other hand, in low level statically typed languages (like C), one must specify that `a` is an integer variable.[4]

In a lot of low level languages (like C and C++), there is a maximum and minimum integer value that an integer variable can hold, but you won't have to worry about them in high level languages.

You can do basic arithmetics with these:

```
a = 1       // a holds 1
b = a + 1   // b holds 2
b = b + 3   // b holds 5
```

### 3.1.2 Float and Double ($\mathbb{R}$)

Consider the following pseudocode.

```
// Dynamically typed languages (eg: Python)
a = 1.5
// Statically typed language (eg: C)
float a = 1.5;
double b = 1.5;
```

This time `a` (and `b`) holds the value 1.5. `a` is a float variable (in both cases), and `b` is a double variable.

One may ask, **what is the distinction between float and double**? Computers obviously cannot be arbitrarily precise, so they must truncate the decimal point at some point.

Floating point numbers are less precise in the way that they keep less number of digits, so in production code, people *usually use double precision* unless that level of precision is not necessary.[5] On the other hand, float-

[4]Some statically typed languages like C++ and rust can actually infer types of the variables at compile time, but you can explicitly assign them.

[5]In high level languages though, they might not have double, and what they mean by "float" already has high enough precision.

ing point numbers have higher range compared to double, so *if the number "varies" a lot, use floating point number.*

One might also ask **why use integers then** if floating or double can represent their values? In the case that we want to enumerate/index from a list:

```
a = [1, 2, 3]
a[0]    // First entry, which is 1
```

integers are obviously the well-suited, unlike float or double (like what is $1.5^{\text{th}}$ number?)

Another quirk of floating point number is that they are not precise even in the simplest computation. Try the following in Python:

```
# Checking if 0.1 + 0.2 is equal to 0.3
0.1 + 0.2 == 0.3
```

Spoiler: The answer is false! Computer will argue that $0.1 + 0.2$ is $0.30000000000000004$; this is due to computers using base-2 rather than base-10.

You can also do arithmetic with them:

```
// Dynamically typed languages
a = 1.5
b = 2
c = a + b   // float variable holding 3.5
// Statically typed languages
float a = 1.5;
int b = 2;
float c = a + b;
```

(Note that if you add an integer and a float (or double), you are likely to get float (or double), unless you explicitly use type casting or otherwise.)

## 3.2 String ($\bigcup_{n=0}^{\infty} \{a, \cdots, z, A, \cdots, Z, 0, \cdots, 9, +, \cdots\}^n$)

Consider the following:

```
// Python
a = 'ABC'
```

`a` is a variable containing a **string**, a collection of letters. Note that I did not include C (and other low level language) version; string is slightly more complicated in those languages.

You can do string concatenation, that is, joining two strings together.

```
a = 'abc'
b = 'defg'
c = a + b   // 'abcdefg'
```

### 3.3 Boolean ($\{T, F\}$)

**Boolean** values represent the truth or false.

```
a = True
b = False
c = a and b     // False
d = a or b      // True
e = (not True)  // False
```

Common operators include `and`, `or`, `xor`, and `not`. They are self-explanatory.

## 4 List and Tuple ($\bigcup_{n=0}^{\infty} A^n$)

Suppose you are trying to store grades of 30 people. You could create the variables for 30 people as such:

```
grade_01 = 100
grade_02 = 59
...
grade_30 = 60
```

Of course, this does not scale if you are trying to write a program that you would like to ship for production, that is, you should have a more "dynamic" way to capture multitude of elements. This is where **list** comes in:

```
grade = [100, 59, ..., 60]
// Accessing the first grade:
print(grade[0])
// Accessing the second grade:
print(grade[1])
// Accessing the last grade:
print(grade[-1])
```

Note that in computer science it is common to count from 0. Accessing the last grade by indexing it with -1 is not really a common thing, but a lot of languages have their own way of supporting it.

Note that the elements of a list all have their own type[6].

It is likely that you can modify element of the list:

```
grade = [100, 59, ..., 60]
// Mistake in the first grade! Should've been 75...
grade[0] = 75
```

You can also add or remove element in a list:

```
grade = [100, 59, 20]
grade.push(68)      // Add at the end of a list
```

[6]Some languages enforce that all the elements should have the same type, but not all

```
grade.pop()          // Remove the element just added.
grade.insert(0, 75) // Insert at the front of the list
                     // the value 75.
grade.remove(1)      // Remove the second element of the list.
```

On the other hand, sometimes we want multitude of data, but do not want the flexibility to modify it. For example, altering the dimension of a frame in a video is not desirable most of the time. We can capture this by **tuple**.

```
dimension = (1920, 1080)
// Access
width = dimension.first
height = dimension.second
```

# 5 Conditional

A program often deals with some sort of logic.

```
a = 19
if (a < 19) {
    print('You are not allowed at the bar!');
}
else if (a < 25) {
    print('Gonna check your ID just in case');
}
else {
    print('Welcome!');
}
```

This is what is known as **if statement**. The nineteen year old would be ID-checked in this program.

In an *if* statement, the code block is executed if the expression passed evaluates to `True`, the boolean value:

```
if (True)
{
    // Whatever is inside here gets executed.
}
```

Variants of *if* statement are *if-else* and *if-elseif-else* statements.

```
// If-else statement
if (STATEMENT)
{
    // Some stuff
}
else
{
```

```
    // Some other stuff
}

// If-elseif-else statement
if (STATEMENT)
{
    // Some stuff
}
else if (STATEMENT)
{
    // Some other stuff
}
else
{
    // Some other other stuff
}
```

In Python, you would use `elif` instead of `else if`, and code blocks are indented rather than enclosed by curly brackets.

Side Note: It is incredibly common that the check-of-equality is done by `==` rather than `=`.

```
a = 1        // a IS 1
a == 1       // IS a 1? (True)
```

# 6 Loops

Computers are designed to do repetitive stuff really well. In order to program in a repetitive stuff, you have to know **loops**.

There are different kinds of loops. Let's see some of the common ones.

## 6.1 While Loop

This is pretty much the god of loops, yet you won't use it unless needed. **While loops** can basically do anything that other loops can do. While loops execute the code in a code block until the condition is false.

```
int i = 0;
while (i < 10)
{
    i = i + 1;      // Repeats ten times (i = 0 ~ 9)
}
```

Note that in the above example, the code block executes as `i` is incremented from 0 to 10. (When `i = 10`, the final check happens, and the code block does not execute.)

You can also do infinite loops:

```
while (True)
{
    // Something
}
```

This is useful for when you want the program to keep going rather than halting.

## 6.2 For Loop

This is a more readible loop. **For loops** often iterate over a variable over some list.

```
// This program prints out all the grades.

grades = [100, 86, 49, 60, 99, 20, 45]

// For Loop Implementation
for grade in grades
{
    print(grade)
}

// While Loop Implementation: Less Readible
i = 0
while i < length(grades)
{
    print(grades[i])
    i = i + 1
}
```

## 6.3 Do While Loop

**Do while loop** is a variant of the while loop. It always executes once, then it checks for the condition later. Useful for when reading a file.

```
// Open a file for reading
file = open('test.txt', 'r')

// Print if we have not reached the end of file
// Read 1024 bytes at each time
do
{
    content = file.read(1024)
    print(content)
} while (file.at() == EOF)
```

```

// Close file
file.close()
```

### 6.4 Break

**Break** is not a type of loop, but rather a feature to get out of a loop in the middle. Consider the machines at a factory. The machine would operate on a loop, but if a factory worker falls on to the machine, it might have to stop immediately. A break could be used to achieve this[7]

```
while (True)
{
    machine.close();
    sleep(5);       // Wait for 5 seconds
    if (personFellIntoIceCreamMachine == True)
    {
        break;
    }
    // Do task
    ...
}
```

## 7 Function

**Function** is a way to reuse code. Consider having a program that utilizes vector dot product a lot. Instead of typing the code for vector dot product every time, one could write a function for vector dot product and "call" it.

```
// Computes dot product in 2D, given two vectors
function dot (v1, v2)
{
    v11 = v1.first
    v12 = v1.second
    v21 = v2.first
    v22 = v2.second
    return v11 * v21 + v12 * v22    // Dot product
}

v1 = (1, 2)
v2 = (5, -2)
print(dot(v1, v2))
```

[7] In reality, accident detection is time critical, so hardware should be designed to cut the power immediately.

Note that **argument of the function** in the definition is not dependent on the variables that are defined outside. For example:

```
x = 2
function addOne (x)
{
    // x is just a temporary variable inside the code block.
    // May not necessarily be 2
    return x + 1
}
print(addOne(4))    // Prints out 5.
```

Note that not only can you enter variables and literals (eg: "hello" or 12) into functions, but you can enter another function into a function argument.

```
// Evaluation map
function eval(f, val)
{
    return f(val)
}
// x -> x + 5
function addFive(x)
{
    return x + 5
}
// x -> 6 * x
function multSix(x)
{
    return 6 * x
}

// Start of execution flow
a = 8
print(eval(addFive, a))     // 8 + 5, so prints 13
print(eval(multSix, a))     // 6 * 8, so prints 48
```

# 8 Class and Object

**Classes** and **objects** are defining features of **OOP (Object Oriented Programming)**. A lot of high level languages such as Python and Java, implement OOP. Obviously this is a gigantic topic, so I won't be able to cover all the details.

## 8.1 Classes

**Class** is a way you can define new data type. Suppose you have a point mass that you want to work with. It may be cleaner to define what a point is and

its functions, and work with that, rather than having to work with bunch of variable names used.

```
// Point3D Class
class Point
{
    // Physical Property
    mass = 1.0;
    // Coordinates
    x = 0.0;
    y = 0.0;
    z = 0.0;

    // Class Function
    // Teleport the point to given coordinate.
    function teleport(ax, ay, az)
    {
        x = ax;
        y = ay;
        z = az;
    }
}
```

## 8.2 Object

Now that you've defined a class, you can instantiate as an object.

```
class Point
{
    ...
}

// Start of execution flow
pt1 = new Point;    // Instantiation
pt1.mass = 3.0;
pt1.x = 1.0;
pt1.y = 2.1;
pt1.z = -3.2;
// Teleport to the origin!
pt1.teleport(0.0, 0.0, 0.0);
```

Note that a lot of OOP allows/requires you to have constructor function for your class; instead of assigning values after instantiation, you assign it during instantiation.

# 9 Lambda Calculus

**Lambda calculus** is the idea that *everything* is a function. Even variables can be thought of as a function in the sense that $v$ is a float variable if it can be thought of as $v \in \{\phi : \emptyset \to \mathbb{R}\}$. This is another gigantic idea, so I won't go into details.

## 9.1 What is $\lambda$?

$\lambda$-calculus is the way of capturing the notion of functions in a systematic way. Consider the syntax:

$$\lambda x.x + 1$$

This represents a function that returns the input plus 1. ($x$ is a binding variable.) What about a function that adds two variables together?

$$\lambda x.\lambda y.x + y$$

If you break this down

$$\lambda x.\lambda y.x + y = \lambda x.\,(\lambda y.x + y)$$

you see that this is actually two functions of the following definitions composed.

$$f : \mathbb{R} \to \{\phi : \mathbb{R} \to \mathbb{R}\}$$
$$g : \mathbb{R} \to \mathbb{R}$$

where $f(x)$ is not a value, but a function such that $(f(x))(y) = x + y$.

Compare the traditional function definition with lambda calculus in action:

```
// Traditional Function
function f1 (x, y)
{
    return x + y;
}

// Lambda Calculus Function
f2 = lambda x, y : x + y
```

You might think that *lambda calculus just a way to implement functions in a simple way.* The truth is, it's more than that. Lambda calculus bypasses the need of having variables, which have *state*. This means you are guaranteed to get the same result if you execute the same line twice, no matter what.

### 9.2 Example: Implementation of Boolean

Let's start from scratch. Suppose you have no data type available other than the lambda syntax. You could "describe" the `True` and `False` as the following:

$$T = \lambda x.\lambda y.x$$
$$F = \lambda x.\lambda y.y$$

## 10 Exercise

**Exercise 1** (Even Numbers)**.** Print every integer from 0 to 99 that is even.

**Exercise 2** (Quadratic Solver)**.** Write a program that solves the quadratic equation of the form

$$ax^2 + bx + c = 0$$

given $a, b, c$. (Be careful when $a = 0$, or when the discriminant is negative.)

**Exercise 3** (Factorial)**.** Write a program that computes factorial $n!$ given $n \in \mathbb{Z}^{\geq 0}$.

**Exercise 4** (Newton's Method)**.** Apply Newton's method to $f(x) = x^2 - 1$ to find the positive root, starting from $x = 1.2$.

**Exercise 5** (Quadrature)**.** Write a **function** that takes in another function (assuming a continuous function over $[-1, 1]$) and outputs its approximate integral.

**Exercise 6** (Eigenvector)**.** Given symmetric matrix $A$, you can compute its eigenvector corresponding to the largest eigenvalue (in terms of its modulus) by[8]

$$x_{k+1} = \frac{Ax_k}{||Ax_k||_2}$$

and see where $x_k$ converges to. Try it on:

$$\begin{pmatrix} 0.5 & -0.3 \\ -0.3 & -0.5 \end{pmatrix}$$

**Exercise 7** (AES Encryption (Harder))**.** Look up the mathematics behind AES encryption. Implement a program that encrypts a file. (**Do not use cryptography program that you write in production** as it is deemed an unsafe thing to do. Even experts do not implement their cryptographic function.)

[8] $||\cdot||_2$ is the Euclidean norm.