

# programming languages - 2

Dr. Tuğberk Kocatekin  
Istanbul Arel University

# Procedural Units (functions, parameters)

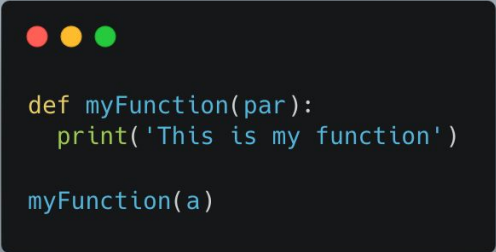
— — —

- Functions
- Dividing a large program into manageable units is advantageous.
- The term **function** is widely used in imperative languages
- **Method** is generally used in object-oriented languages.

# Function

---

- Set of instructions for performing a task that can be used by other program units. Control is transferred to the function and returned to original later.
- Functions are **called** or **invoked**.

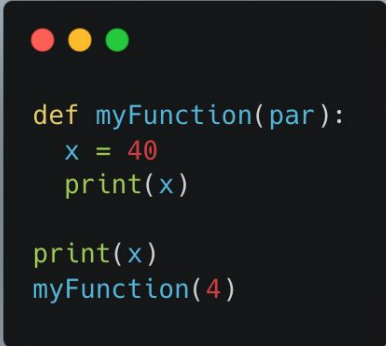


```
def myFunction(par):  
    print('This is my function')  
  
myFunction(a)
```

# Function

— — —

- A *variable* declared in the function is a **local variable**.
- It means that, outside the function you cannot reach that variable.
- Therefore, two functions can have variables with the same name.
- The portion of the program in which a variable can be referenced is called the **scope**.
- The scope of a local variable is the function in which it is declared.
- Variables whose scopes are not **restricted** are called **global variables**.



```
def myFunction(par):  
    x = 40  
    print(x)  
  
print(x)  
myFunction(4)
```

# Parameters

— — —

- Generic terms within the functions are called **parameters**.
- Formal parameters
  - Terms used within the function.
- Actual parameters
  - Precise meanings assigned to these formal parameters when the function is applied.
- Void functions: In some languages you need to declare the type of your return. If you use **void**, it means you don't have to return anything. It is different from language to language.

# Actual and formal parameters

— — —

- Formal parameters are the parameters in the function.
- Actual parameters are what you are passing into a function.

# Pass by Value / Reference

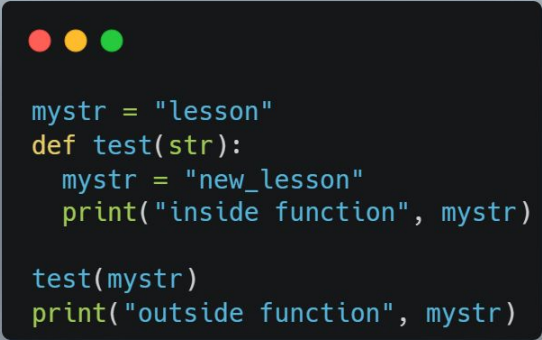
— — —

- Pass by value
  - Function is called by directly passing the value of the variable as an argument. Any changes made inside the function **does not affect the original value.**
  - Parameters passed as an arguments create its own copy. So, the changes made inside the function are made to the copied value, not the original.
- Pass by reference
  - Function is called by directly passing the reference(address) of the variable as an argument. So, changing the value inside the function also **changes the original value.**

# Pass by Value

---

- Here, we see that the variable *mystr* will show different results even if it is changed in the function.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains Python code that demonstrates the 'pass by value' concept. The code defines a function 'test' that takes a string parameter 'str' and assigns it to a local variable 'mystr'. Inside the function, 'mystr' is reassigned to 'new\_lesson' and a message is printed. Outside the function, 'test(mystr)' is called with the current value of 'mystr' ('lesson'), and another message is printed. This shows that the function's changes to 'mystr' do not affect the variable outside the function.

```
mystr = "lesson"
def test(str):
    mystr = "new_lesson"
    print("inside function", mystr)

test(mystr)
print("outside function", mystr)
```



# Pass by reference

— — —

- We change the value inside the address holding the value.
- When we use *references* in formal parameters, it is called pass by reference.
- Modern languages are *pass by value*.

```
#include <iostream>

using namespace std;

void Twice(int&, int&);           // DECLARE before use

int main()
{
    int x = 5, y = 8;

    cout << "Initial values of variables:\n";
    cout << "\tx = ";
    cout << x;
    cout << "\ty = ";
    cout << y;
    cout << '\n';

    cout << "Calling the function Twice(x,y)\n";

    Twice(x,y);

    cout << "The new values of x and y are:\n";
    cout << "\tx = " << x << "\ty = " << y << '\n';
    cout << "Goodbye!\n";
}

void Twice(int& a, int& b)
{
    a *= 2;
    b *= 2;
}
```

# Fruitful functions

---

- Functions which return a value.
- Different languages declare functions in different ways. For example in C++ or Java, you declare the type of the return value while declaring the function.
  - int, void, etc.
- When we use **void**, it means it does not return any value.
- When you want to store the return value, you can do:
  - `x = sum(4,5)`



```
def sum(a,b):  
    return a+b  
  
x = sum(3,5)  
print(x)
```

# Language Implementation

— — —

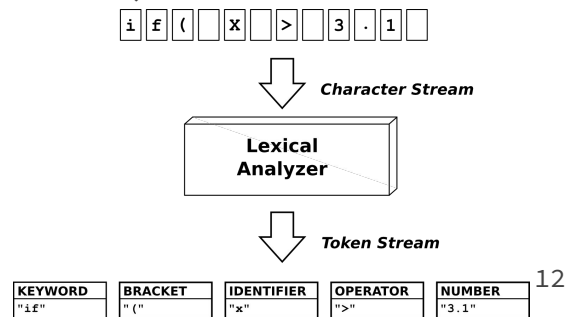
- Translation
  - Process of converting a program from one language to another.
  - Original is called *source program*.
  - Translated version is called *object program*.
  - Consists of 3 activities:
    - Lexical analyzer
    - Parser
    - Code generator

# Lexical Analysis

---

- Lexical analysis

- Process of recognizing which strings or symbols from the source program represents a single *entity* or **token**. For example, 153 is not 1,5,3 it is 153, a single numeric value. Or a *String* such as “computer” is not c,o,.. it is a single unit.
- Lexical analyzer reads the source symbol by symbol and identify groups that represent tokens, classifies those tokens accordingly (numeric values, words, arithmetic operations, etc..)



# Parser (syntax analysis)

---

- Parser
  - Views the program in terms of lexical units (tokens) rather than symbols. Parser joins these group into *statements*. **Parsing is the process of identifying the grammatical structure of the program and recognizing the role of each component.**
- There are **keywords** in programming languages, these are *reserved words*. (if, then, else, etc.) They also use most languages use semicolons, punctuation marks to help parsers.
- The parsing process is based on a set of rules that define the *syntax* of the programming language. These rules are called a **grammar**.

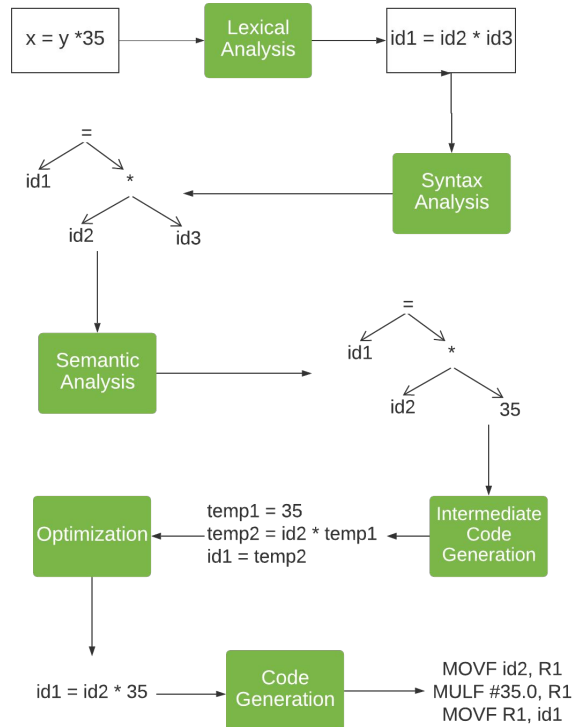
# Code generator

— — —

- Final activity of the translation process.
- Constructs machine-language instructions to implement the statements recognized by the parser.

# Compilers

Compilation Flow Example



# OOP: Object oriented Programming



# Classes and Objects

— — —

- Object
  - An instance of a *class*.
  - Allocates memory space when they are created.
- Class
  - A template for a collection of objects. By this template, objects are created.
  - Does not allocate memory when it is created.

# Keywords

— — —

- **Encapsulation**
  - Bundling data with the methods that operate on that data, so that it is restricted direct access from others. It is used to hide the values or state of an object. Prevents direct access.
  - Public, private.
  - It is also possible in non-OOP languages.
- **Abstraction**
  - Displaying only essential information and hiding the details.
  - You know that pushing the gas pedal accelerates the car but don't know details on how it does that.
- **Polymorphism**
  - Allows to perform a single action in different ways.
- **Inheritance**
  - Allows classes to inherit features of other classes.

# Encapsulation

---

- By using *private* we encapsulate *age* data.
- By using *getter* and *setter* functions, we can reach those data.
- However, it is up to us whether to put them in *get*, *set* functions.

```
class Student{  
    private:  
        int age;  
  
    public:  
        void setData(int studentAge){  
            age = studentAge;  
        }  
  
        int getData() {  
            return age;  
        }  
};
```

# Inheritance

— — —

- When we create a class, instead of writing every method inside it again, we can *inherit* those methods by using a derived class.

```
class Animal {  
    public:  
    void eat() {  
        cout << "I can eat!" << endl;  
    }  
  
    void sleep() {  
        cout << "I can sleep!" << endl;  
    }  
};
```

Base Class

```
class Dog : public Animal {  
    public:  
    void bark() {  
        cout << "Woof woof!!" << endl;  
    }  
};
```

Derived Class

# Abstraction

— — —

- Not the same with encapsulation.
- Here, trivial or non-essential units are not displayed to the user.

```
class AbstractionExample{  
    private:  
        int num;  
        char ch;  
  
    public:  
        void setMyValues(int n, char c) {  
            num = n; ch = c;  
        }  
  
        void getMyValues() {  
            cout<<"Numbers is: "<<num<< endl;  
            cout<<"Char is: "<<ch<<endl;  
        }  
};
```

# Polymorphism

---

- There are types of it. For example in Java, you can do it in two ways:
  - Static / Compile-Time
    - Method overloading
  - Dynamic
    - Method overriding

# Method overloading

— — —

- Use the same method with different parameters and get different results.

```
class Shapes {  
    public void area() {  
        System.out.println("Find area ");  
    }  
    public void area(int r) {  
        System.out.println("Circle area = "+3.14*r*r);  
    }  
  
    public void area(double b, double h) {  
        System.out.println("Triangle area="+0.5*b*h);  
    }  
    public void area(int l, int b) {  
        System.out.println("Rectangle area="+l*b);  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Shapes myShape = new Shapes(); // Create a Shapes object  
  
        myShape.area();  
        myShape.area(5);  
        myShape.area(6.0,1.2);  
        myShape.area(6,2);  
    }  
}
```

# Method overriding

— — —

- Here, Car2 inherits Vehicle class.

```
class Vehicle{
    //defining a method
    void run(){System.out.println("Vehicle is moving");}
}
//Creating a child class
class Car2 extends Vehicle{
    //defining the same method as in the parent class
    void run(){System.out.println("car is running safely");}

    public static void main(String args[]){
        Car2 obj = new Car2();//creating object
        obj.run();//calling method
    }
}
```