Subprograms

Lecture 15

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Introduction

- Two fundamental abstraction facilities can be included in a programming language:
  - Process abstraction
  - Data abstraction

- In the early history of high-level programming languages, only process abstraction was included.

- Process abstraction, in the form of subprograms, has been a central concept in all programming languages.

Fundamentals of Subprograms: General Subprogram Characteristics

- All subprograms discussed have the following characteristics:
  - Each subprogram has a single entry point.
  - The calling program unit is suspended during the execution of the called subprogram, which implies that there is only one subprogram in execution at any given time.
  - Control always returns to the caller when the subprogram execution terminates.

Fundamentals of Subprograms: Basic Definitions

- A ***subprogram definition*** describes the *interface* and the *actions* of the subprogram abstraction.
- A ***subprogram call*** is the explicit request that a specific subprogram be executed.
- A subprogram is said to be ***active*** if, after having been called, it has begun execution but has not yet completed that execution.

Fundamentals of Subprograms: Basic Definitions

- A ***subprogram header***, which is the first part of the *definition*, serves several purposes.
  - First, it specifies that the following syntactic unit is a subprogram definition of some particular kind.
  - Second, the header provides a name for the subprogram.
  - Third, it may optionally specify a list of parameters.

Fundamentals of Subprograms: Basic Definitions

- Consider the following header examples:

```
def adder (parameters):
```

  - This is the header of a Python subprogram named adder.
  - Ruby subprogram headers also begin with ***def***.

```
def calculate_value(x, y)
        x + y
end
```

Fundamentals of Subprograms: Basic Definitions

- Consider the following header examples:

  def adder (parameters):

  - This is the header of a Python subprogram named adder.
  - Ruby subprogram headers also begin with ***def***.
  - The header of a JavaScript subprogram begins with ***function***.

```
function name (argument1, argument2, ...)

function AbsoluteValue (x) {
    if (x < 0) {
        x = -x;
    }
    return x;
}
```

Fundamentals of Subprograms: Basic Definitions

- In C, the header of a function named adder might be as follows:

  void adder (parameters)

  - The reserved word ***void*** in this header indicates that the subprogram does not return a value.

```
int adder (int a, int b) {
        return a + b;
}
```

Fundamentals of Subprograms: Basic Definitions

- The *body* of subprograms defines its *actions*.
- In the C-based languages (and some others—for example, JavaScript) the body of a subprogram is delimited by ***braces***.
- In Ruby, an ***end*** statement terminates the body of a subprogram.
- As with compound statements, the statements in the body of a Python function must be ***indented*** and the end of the body is indicated by the first statement that is not indented.

Fundamentals of Subprograms: Basic Definitions

- One characteristic of Python functions that sets them apart from the functions of other common programming languages is that function ***def*** statements are executable.

- When a ***def*** statement is executed, it assigns the given name to the given function body.

- Until a function's ***def*** has been executed, the function cannot be called.

Fundamentals of Subprograms: Basic Definitions

- Consider the following skeletal example:

```
if . . .
   def fun1(. . .):
       . . .
else
   def fun2(. . .):
       . . .
```

- A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function).
- The function definition does not execute the function body; this gets executed only when the function is called.

Fundamentals of Subprograms: Basic Definitions

- Subprograms can have ***declarations*** as well as ***definitions***.

- This form parallels the *variable declarations* and definitions in C, in which the declarations can be used to provide type information but not to define variables.

- Subprogram declarations provide the subprogram's ***protocol*** but do not include their ***bodies***.

Fundamentals of Subprograms: Basic Definitions

- In both the cases of variables and subprograms, declarations are needed for static type checking.
  - In the case of subprograms, it is the type of the parameters that must be checked.

- Function declarations are common in C and C++ programs, where they are called ***prototypes***.
  - Such declarations are often placed in *header files*.

Fundamentals of Subprograms: Basic Definitions

- Example of ***function declaration***:

  int *max* (***int*** *a*, ***int*** *b*);

- Example of ***function definition***:

```
int max(int a, int b) {
    /* local variable declaration */
    int result;
    if (a > b)
        result = a;
    else
        result = b;
    return result;
}
```

Fundamentals of Subprograms: Parameters

- Subprograms typically describe computations.

- There are two ways that a subprogram can gain access to the data that it is to process:
  - *through direct access to nonlocal variables* (declared elsewhere but visible in the subprogram)
  - *through parameter passing*

Fundamentals of Subprograms: Parameters

- Subprograms typically describe computations.

- There are two ways that a subprogram can gain access to the data that it is to process:
  - *through direct access to nonlocal variables* (declared elsewhere but visible in the subprogram)

```
x = "global"
def foo():
    print("x inside:", x)

foo()
print("x outside:", x)
```

Fundamentals of Subprograms: Parameters

- Subprograms typically describe computations.
- There are two ways that a subprogram can gain access to the data that it is to process:
  - *through parameter passing*

```
def func1(list):
   print list
   list = [47,11]
   print list

fib = [0,1,1,2,3,5,8]
func1(fib)
```

Fundamentals of Subprograms: Parameters

- Data passed through parameters are accessed through names that are local to the subprogram.
- Parameter passing is more flexible than direct access to nonlocal variables.
- In essence, a subprogram with parameter access to the data that it is to process is a *parameterized computation*.
- It can perform its computation on whatever data it receives through its parameters (presuming the types of the parameters are as expected by the subprogram).

Fundamentals of Subprograms: Parameters

- If data access is through nonlocal variables, the only way the computation can proceed on different data is to assign new values to those nonlocal variables between calls to the subprogram.
- Extensive access to nonlocals can reduce reliability.
- Variables that are visible to the subprogram where access is desired often end up also being visible where access to them is not needed.

Fundamentals of Subprograms: Parameters

- The parameters in the ***subprogram header*** are called ***formal parameters***.

```
int adder (int a, int b) {
    return a + b;
}
```

- They are sometimes thought of as ***dummy variables*** because they are not variables in the usual sense:
  - In most cases, they are bound to storage only when the subprogram is called, and that binding is often through some other program variables.

Fundamentals of Subprograms: Parameters

- ***Subprogram call*** statements must include the name of the subprogram and a list of parameters to be bound to the *formal parameters* of the subprogram.

- These parameters are called ***actual parameters***.

  sum = adder (***x***, ***y***);

- They must be distinguished from *formal parameters*, because the two usually have different restrictions on their forms, and of course, their uses are quite different.

Fundamentals of Subprograms: Parameters

- In nearly all programming languages, the correspondence between ***actual*** and ***formal*** parameters—or the binding of ***actual*** parameters to ***formal*** parameters—is done by ***position***:
  - The ***first actual parameter*** is bound to the ***first formal parameter*** and so forth.
  - Such parameters are called ***positional parameters***.
  - This is an effective and safe method of relating actual parameters to their corresponding formal parameters, as long as the parameter lists are relatively short.

Fundamentals of Subprograms: Parameters

- When lists are long, however, it is easy for a programmer to make mistakes in ***the order of actual parameters*** in the list.

```
def myFunction(alpha, beta, gamma, zeta, alphaList, betaList, gammaList, zetaList):
    …
```

- One solution to this problem is to provide ***keyword parameters***
  - ***The name of the formal parameter*** to which an actual parameter is to be bound is specified with the actual parameter in a call.

- The advantage of ***keyword parameters*** is that they can appear in any order in the actual parameter list.

Fundamentals of Subprograms: Parameters

- Python functions can be called using this technique, as in

  sumer (***length*** = my_length, ***list*** = my_array, ***sum*** = my_sum)

  - where the definition of ***sumer*** has the *formal parameters* ***length***, ***list***, and ***sum***.
  - The disadvantage to keyword parameters is that the user of the subprogram must know the names of formal parameters.

Fundamentals of Subprograms: Parameters

- In addition to keyword parameters, Ada, Fortran 95+ and Python allow positional parameters.
- Keyword and positional parameters can be mixed in a call, as in

sumer (my_length, ***sum*** = my_sum, ***list*** = my_array)

  - The only restriction with this approach is that after a positional parameter appears in the list, all remaining parameters must be keyworded.
  - This restriction is necessary because a position may no longer be well defined after a keyword parameter has appeared.

Fundamentals of Subprograms: Parameters

- In Python, Ruby, C++, Fortran 95+ Ada, and PHP, ***formal parameters*** can have ***default values***.
  - A default value is used if no actual parameter is passed to the formal parameter in the subprogram header.

- Consider the following Python function header:

def compute_pay (income, exemptions = 1, tax_rate)

pay = compute_pay (20000.0, tax_rate = 0.15)

Fundamentals of Subprograms: Parameters

- In C++, which does not support keyword parameters, the rules for default parameters are necessarily different.
- The default parameters must ***appear last***, because parameters are positionally associated.
- Once a default parameter is omitted in a call, all remaining formal parameters must have default values.

```
float compute_pay (float income, float tax_rate, int exemptions = 1)

pay = compute_pay(20000.0, 0.15);
```