Learning Aims

- Be familiar with subroutines (functions and procedures), their uses and advantages
- Use subroutines that return values to the calling routine
- Describe the use of parameters to pass data to subroutines by value and by reference
- Contrast the use of local and global variables

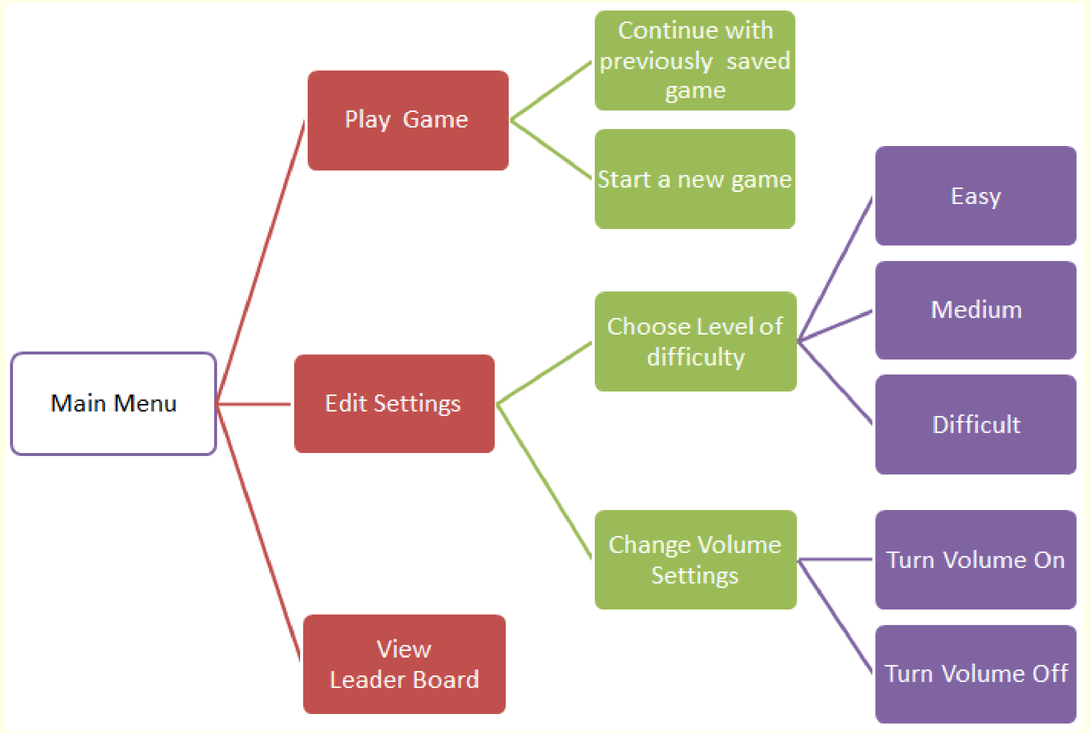
Key terms

Blocks of Code	**Passing** Data
Modularity Function Procedure Subroutine	Argument Parameter

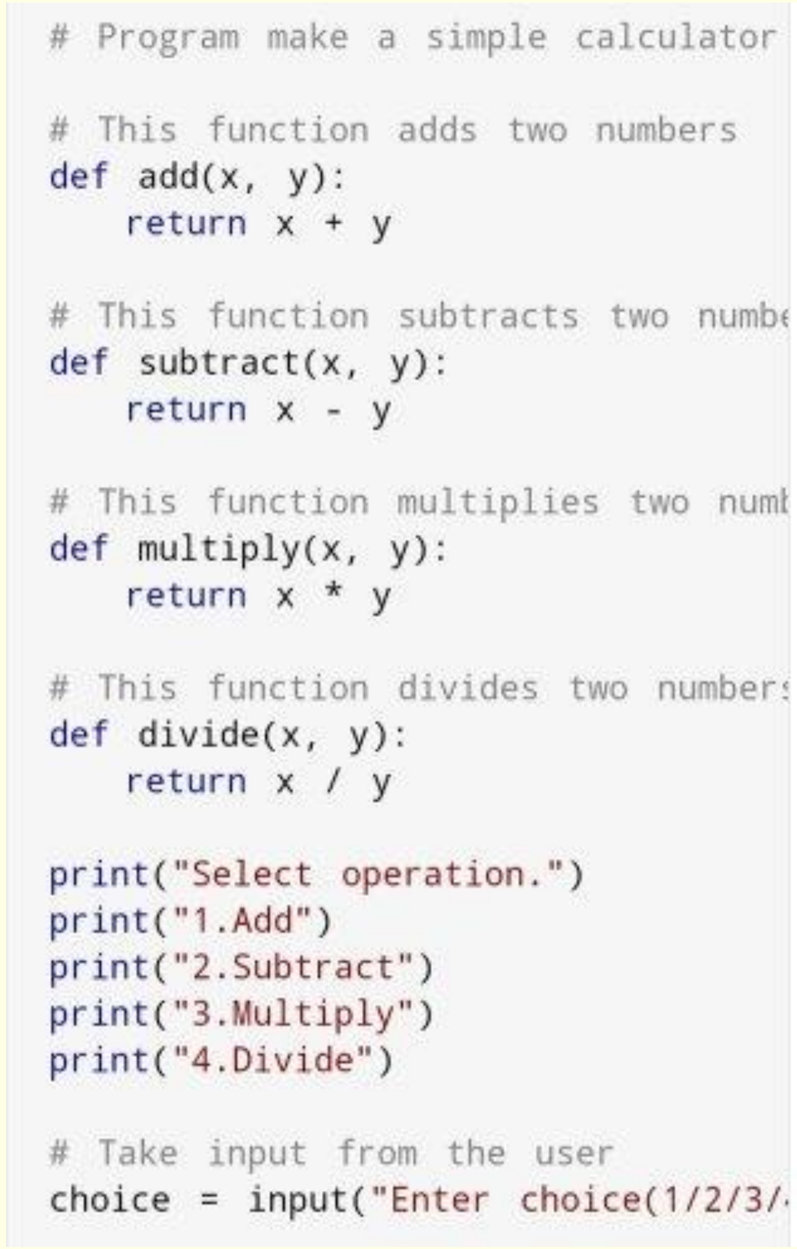
Passing data to subroutine
Calling a subroutine

Why is it good to develop code in a modular way?

- Modularity is used to break a program down into manageable parts that can be self-contained and tested independently.
- It allows a team of programmers to split tasks and focus on specific parts of the program.
- Each module carries out a single, specific task.
- Used to reuse code
- To save you rewriting lots of code again and again you might use a sub routine from an existing library
- It allows faster program development as it allows programmers to re-use previously created code, so we don't have to repeat it.



Structuring code



```
# Program make a simple calculator

# This function adds two numbers
def add(x, y):
    return x + y

# This function subtracts two numbe
def subtract(x, y):
    return x - y

# This function multiplies two numb
def multiply(x, y):
    return x * y

# This function divides two numbers
def divide(x, y):
    return x / y

print("Select operation.")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")

# Take input from the user
choice = input("Enter choice(1/2/3/
```

Procedures vs Functions

Procedure

- Performs a set task
- Takes in zero, one or more parameters

Function

- Performs a set task
- Takes in zero, one or more parameters
- **Returns a value**

Procedure Syntax

```
procedure procedure_name(parameters):
    instructions
end procedure
```

Python example

```
def result(name, age, gender):
    print (“My name is” +name)
    print (“My age is” + str(age))
    print(“My gender is” + gender)

result(“Your name”, 13, “gender”)
```

Parameters

Arguments

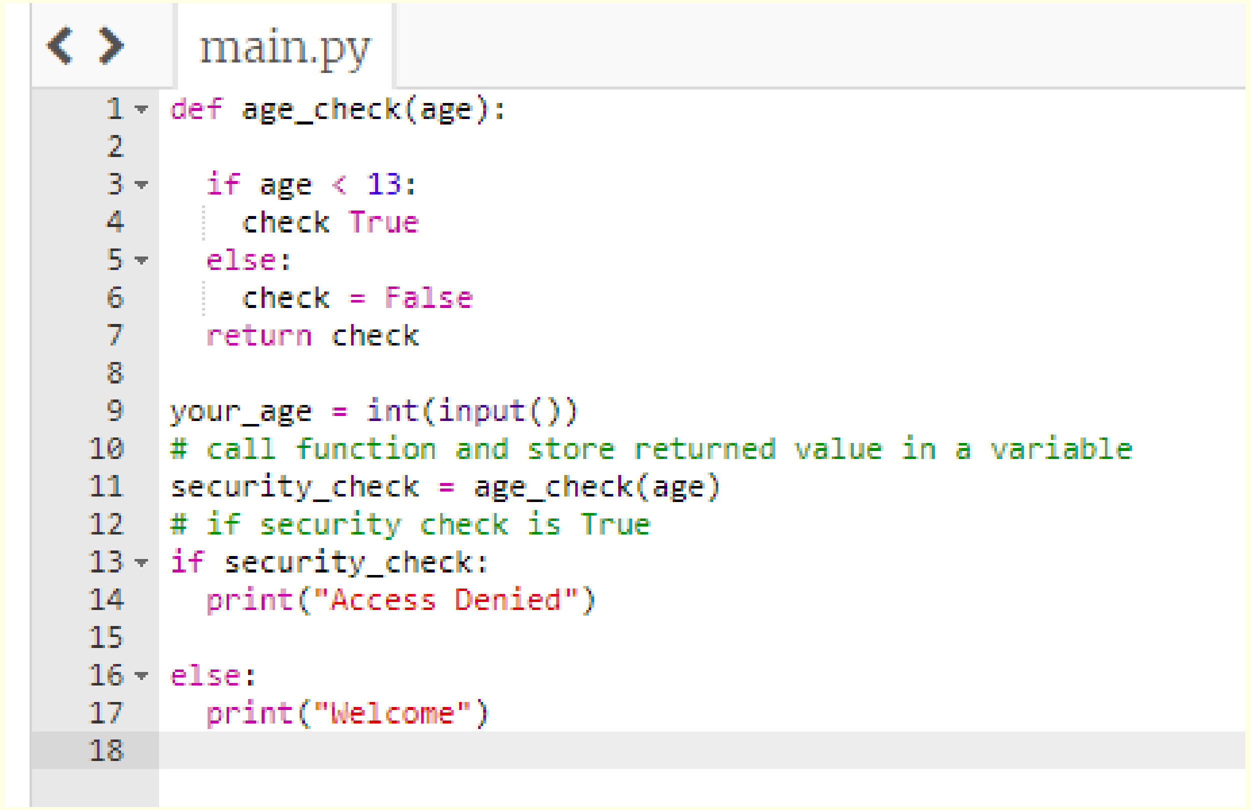
Parameter is variable in the declaration of procedure/function.

Argument is the actual value of this variable that gets passed to function.

Function Syntax

```
function function_name(parameters):
    instructions
    return variable/ list
end function
```

The return statement is used to specify the output.


main.py

```python
def age_check(age):

  if age < 13:
    check True
  else:
    check = False
  return check

your_age = int(input())
# call function and store returned value in a variable
security_check = age_check(age)
# if security check is True
if security_check:
  print("Access Denied")

else:
  print("Welcome")

```

The returned value will need to be stored in a variable

Advantages of using Subroutines

- **Breaking down** or **decomposing** a complex programming task into **smaller sub-tasks** and **writing** each of these as subroutines, makes the **problem** easier to solve.
- **Subroutines** can be used several times within a program.
- It saves the programmer time as it **reduces the amount of code** that needs to be written or amended by allowing you to reuse code **without** having to write it again.
- If you are working as part of a team you can divide a large program into smaller sections and allow individuals to simultaneously work on those sections.
- It makes the code easier to read if you use sensible subroutine labels as the headings tell the reader what that section of code is doing.
- By **reducing** the amount of **repeating tasks** you also **reduce the risk** of introducing errors in a program.
- **Easy to maintain** as each subroutine can be tested separately.

Flowcharting a subprogram

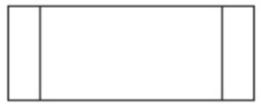


Figure 1.7.3 The symbol for a subprogram in a flowchart

Flowchart

In a flowchart, a subprogram is represented by the symbol shown in Figure 1.7.3.

Fully documenting a complex solution using flowcharts can be cumbersome, and it's quite easy to get lost. Figure 1.7.4 shows what part of the noughts and crosses game might look like using subprograms.

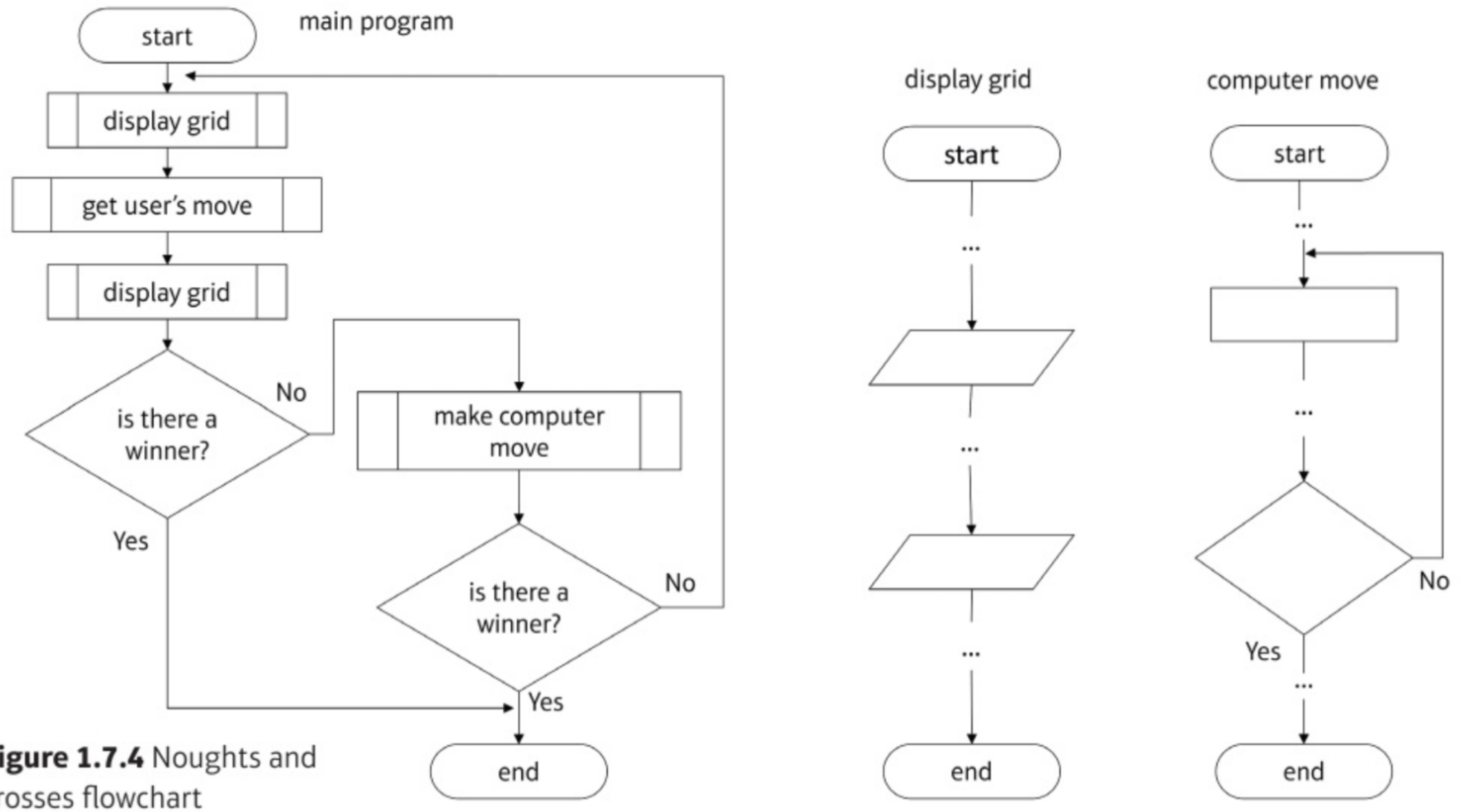
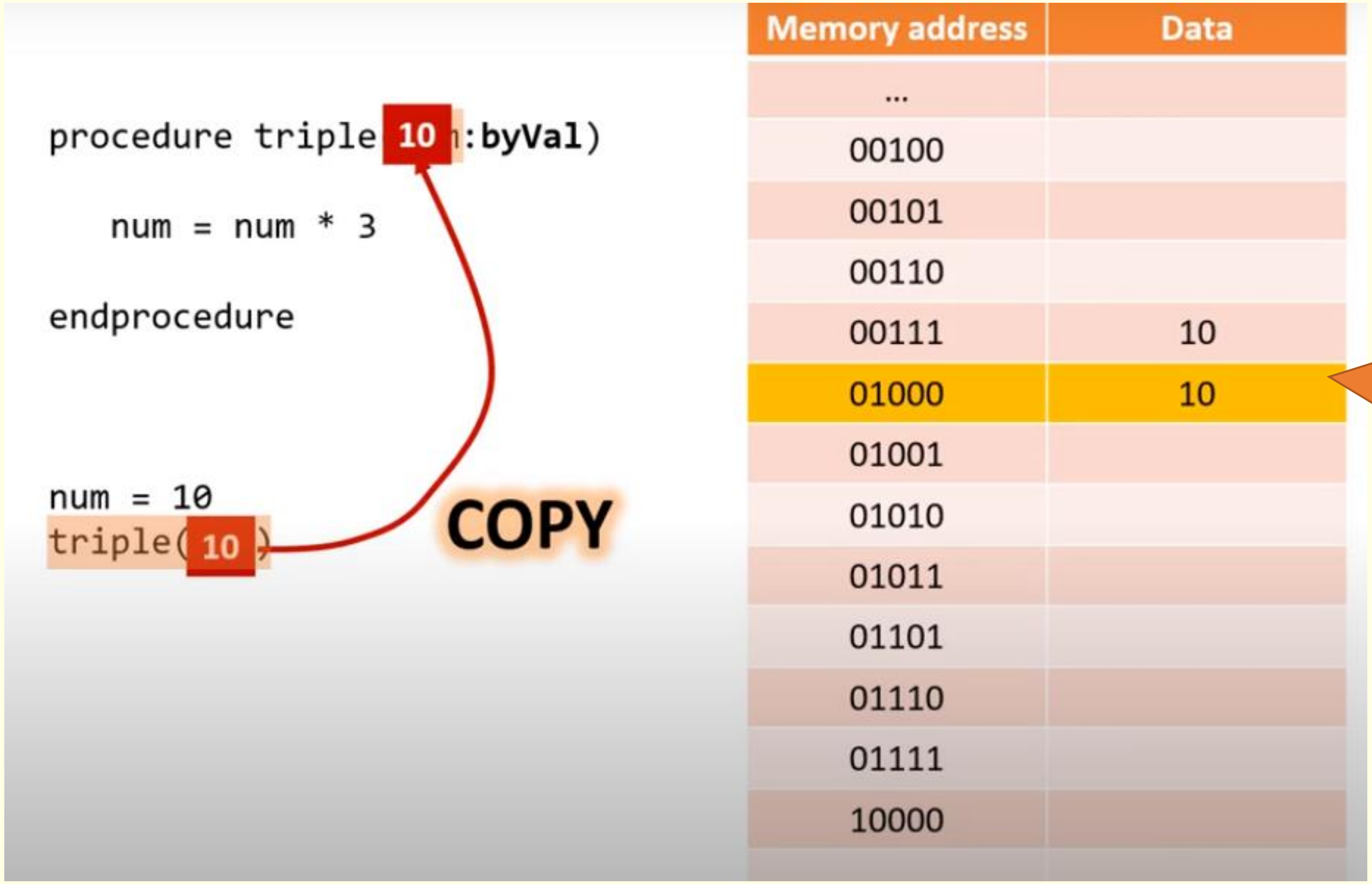


Figure 1.7.4 Noughts and crosses flowchart

Types of Parameters:

- When parameters/arguments are passed to a procedure/function they can be passed in two ways:
- Passing by **value -** a **copy** of the original data is passed to the function and any changes made are lost as soon as the function is no longer in use.
- Passing by **reference -** the function receives a pointer to the actual **memory address** where the data is stored. This means that the function works directly with the original data and if it changes, it stays changed.

Example: Passing by Value

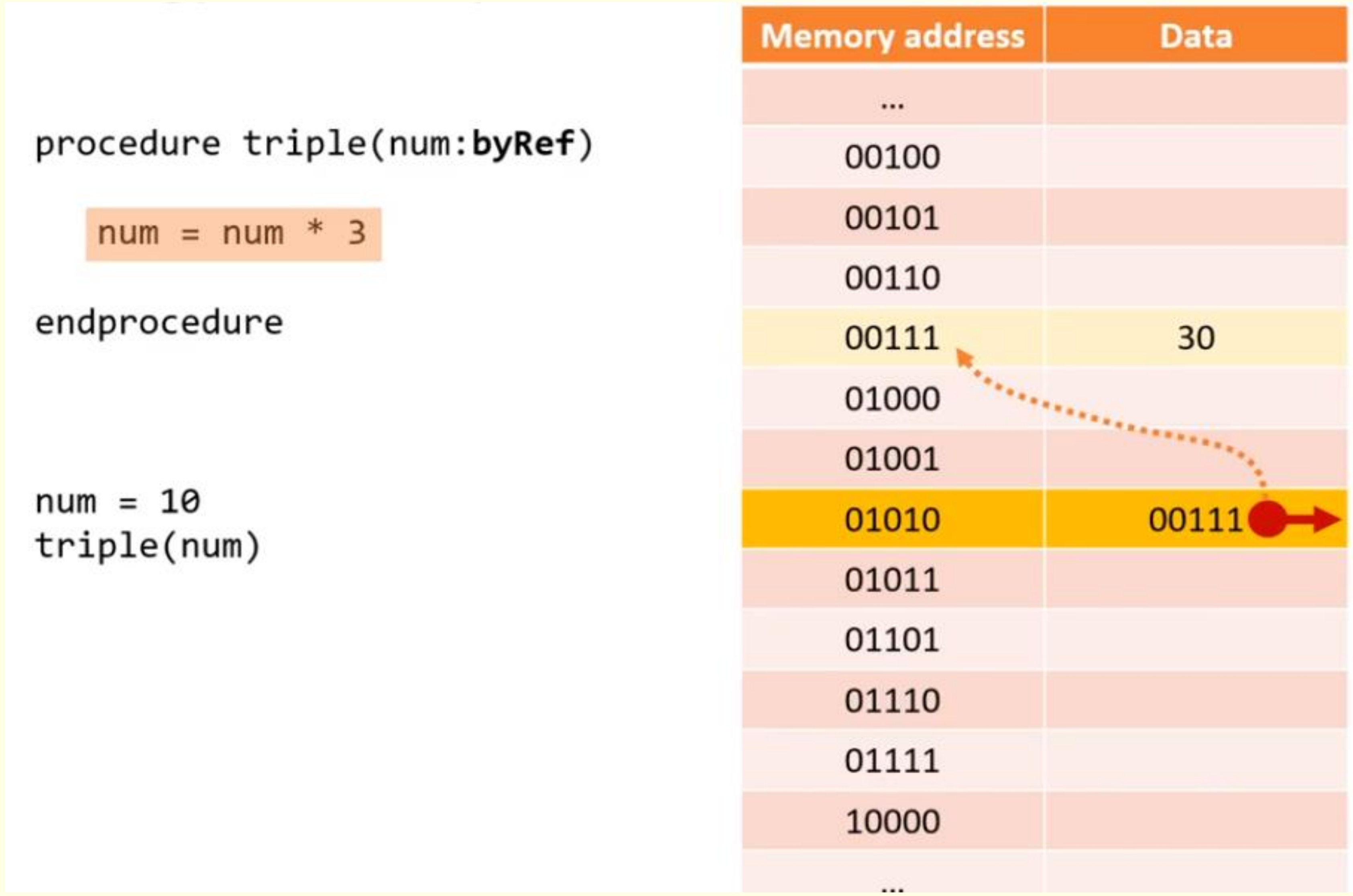


```
procedure triple(10 :byVal)
    num = num * 3
endprocedure


num = 10
triple(10)
```

Memory address	Data
...	
00100	
00101	
00110	
00111	10
01000	10
01001	
01010	
01011	
01101	
01110	
01111	
10000	

A copy of the original data

Example: Passing by Reference

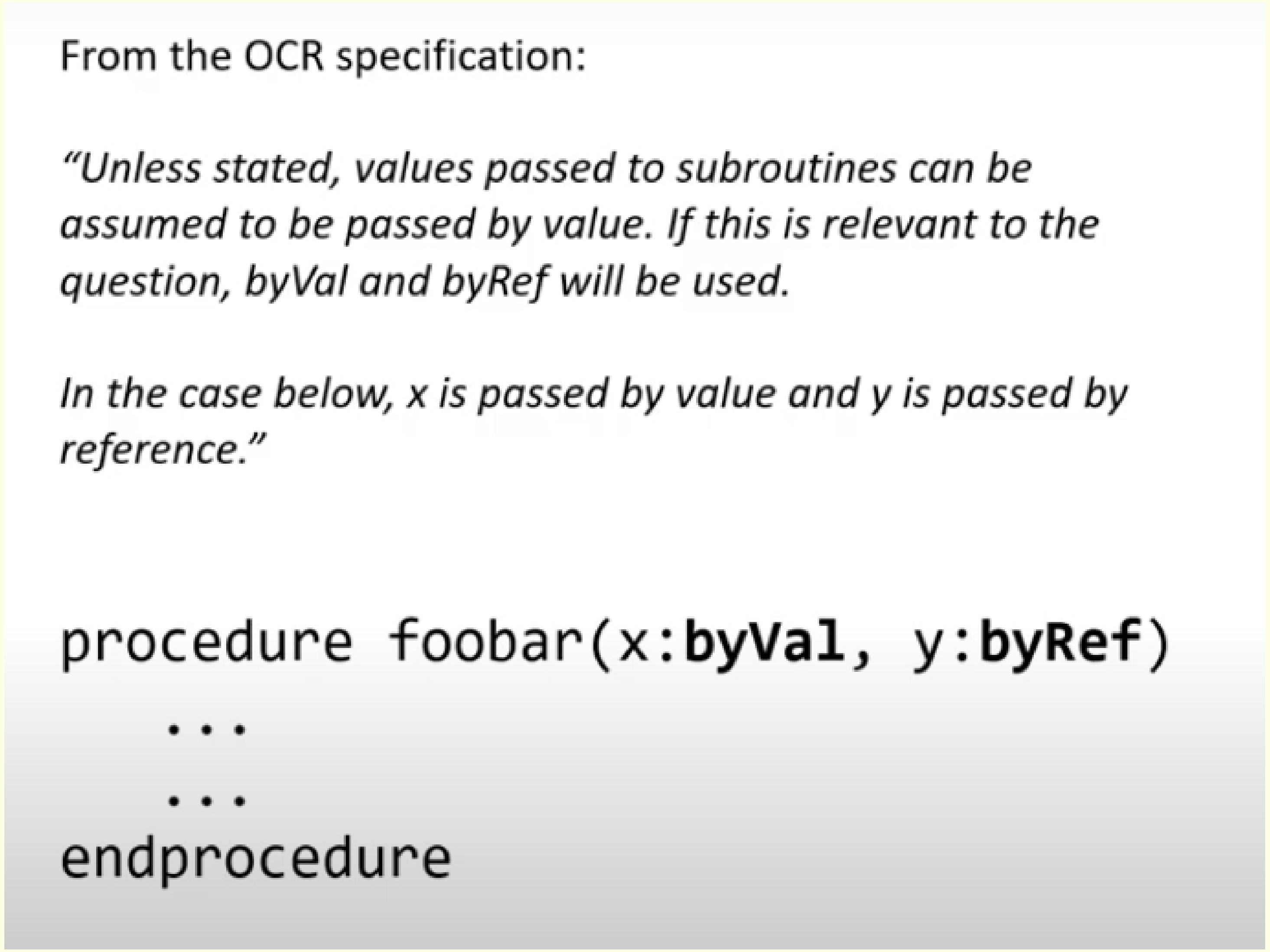


```
procedure triple(num:byRef)

    num = num * 3

endprocedure



num = 10
triple(num)
```

Memory address	Data
...	
00100	
00101	
00110	
00111	30
01000	
01001	
01010	00111
01011	
01101	
01110	
01111	
10000	
...	

Overwriting the original data

Exam Top Tips



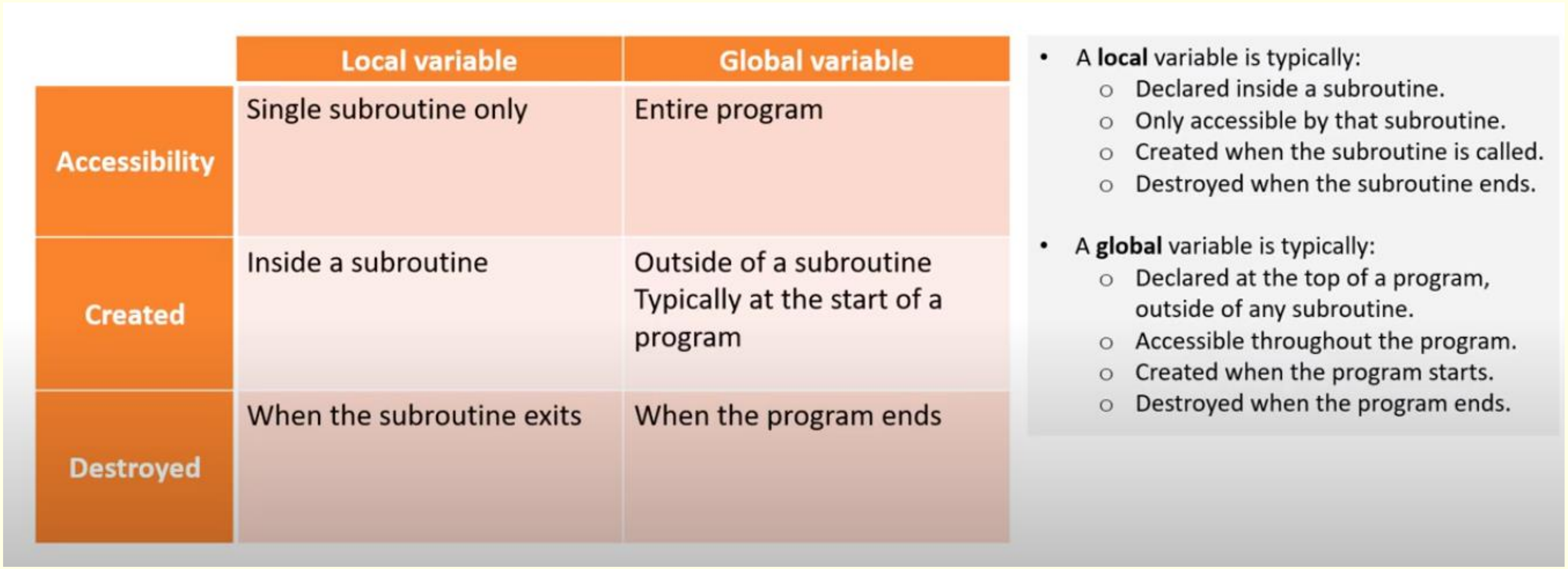
From the OCR specification:

"Unless stated, values passed to subroutines can be assumed to be passed by value. If this is relevant to the question, byVal and byRef will be used.

In the case below, x is passed by value and y is passed by reference."

```
procedure foobar(x:byVal, y:byRef)
    ...
    ...
endprocedure
```

Variable Scope – When declaring a variable or constant the programmer needs to aware of its scope – global or local



	Local variable	Global variable
Accessibility	Single subroutine only	Entire program
Created	Inside a subroutine	Outside of a subroutine Typically at the start of a program
Destroyed	When the subroutine exits	When the program ends

- A **local** variable is typically:
  - Declared inside a subroutine.
  - Only accessible by that subroutine.
  - Created when the subroutine is called.
  - Destroyed when the subroutine ends.
- A **global** variable is typically:
  - Declared at the top of a program, outside of any subroutine.
  - Accessible throughout the program.
  - Created when the program starts.
  - Destroyed when the program ends.

Global & Local Variables in Python

```
#local variable
def foo():
        y = "local"
        print(y)

foo()
```

```
#assigning and accessing a global variable
x = "global“
print(x)
def foo():

        global x
        y = "local"
        x = x * 2
        print(x)
        print(y)
foo()
print(x)
print(y)
```

Good Practice – Avoid global variables

- Good practice is to avoid the use of global variables in favour of local variables.
- Excessive, unnecessary use of global variables can make programs hard to test, debug and maintain.
- Global variables are more likely to be accidental changed, especially in a larger program when hundreds or thousands of variables are in use.
- A global variable places greater demand on system resources - it needs to be retained in memory (RAM) throughout the entire execution of a program whereas the values of local variables will be held in RAM only during the execution of this sub-program.
- Local variables should be used where possible for this reason and also as they use less memory as they are created and destroyed within their subroutine.

A better approach is **Parameter Passing:**

```
maximum = 100.00

def postage(cost, max):

   print("Free postage if you spend more than "
+str(maximum))
   if cost < max:
     cost = cost + 3.50
   print( "£"+ " " + str(cost))


totalAmount = float(input("Enter Total"))
postage (totalAmount, maximum)
```