THE UNIVERSITY OF WESTERN AUSTRALIA

# Lecture 9
# Functions Part 1

## Objectives

- To understand why programs are divided into sets of cooperating functions.
- To be able to define new functions in Python and write programs that use functions.
- To understand the details of function calls and parameter passing in Python.
- To learn how to pass multiple parameters to functions and get (return) multiple results from functions.
- To understand the relationship between actual and formal parameters.

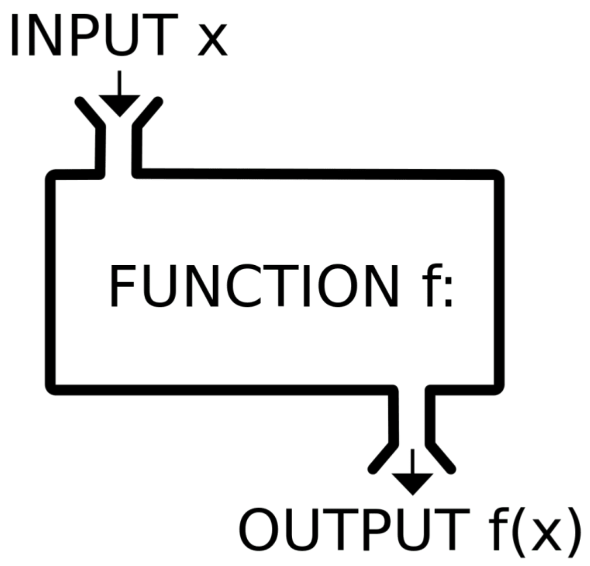
## Why use Functions?

- Having similar code in more than one place has some drawbacks.
  - *Having to type the same code twice or more.*
  - *Unnecessarily complicate the code*
  - *This same code must be maintained in multiple places. Will differ over time as code maintained*

- Functions are used to:
  - *avoid/reduce code duplication*
  - *make programs easy to understand*
  - *make programs easy to maintain.*

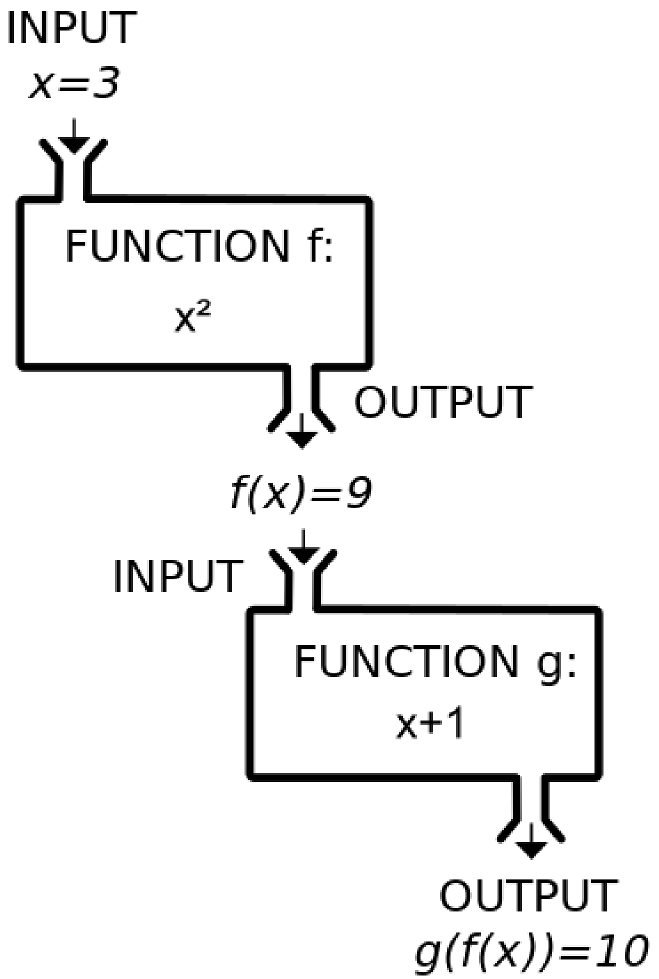
## Functions, Informally

- A function is like a subprogram, a small program inside a program.
- The basic idea – we write a sequence of statements and then give that sequence a name. We can then execute this sequence at any time by referring to the name.
- The part of the program that creates a function is called a function definition.
- When the function is used in a program, we say the definition is called or invoked.



# Functions, Informally

- Multiple functions can be called to do a task
- A problem can be split into multiple parts and each part can be solved by a different team/programmer
- Multiple programmers can work together on bigger projects and share their work in the form of functions



## Functions, Informally

- Happy Birthday lyrics…

```
def main():
    print("Happy birthday to you!" )
    print("Happy birthday to you!" )
    print("Happy birthday, dear Fred...")
    print("Happy birthday to you!")
```

- Gives us this…

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred...
Happy birthday to you!
```

## Functions, Informally

- There's some duplicated code in the program!

```
print("Happy birthday to you!")
```

- We can define a function to print out this line:

```
def happy():
  print("Happy birthday to you!")
```

- With this function, we can rewrite our program.

## Functions, Informally

- The new program –

```
def singFred():
    happy()
    happy()
    print("Happy birthday, dear Fred...")
    happy()
```

- Gives us this output –

```
>>> singFred()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred...
Happy birthday to you!
```

# Functions, Informally

- Creating this function saved us a lot of typing!
- What if it's Lucy's birthday? We could write a new `singLucy` function!

```
def singLucy():
    happy()
    happy()
    print("Happy birthday, dear Lucy...")
    happy()
```

# Functions, Informally

- We could write a main program to sing to both Lucy and Fred

```
def main():
    singFred()
    print()
    singLucy()
```

- This gives us this new output

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred..
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy...
Happy birthday to you!
```

- The only difference is the name in the third print statement.
- These two routines could be collapsed together by using a parameter.

## Functions, Informally

- The generic function *sing*

```
def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + ".")
    happy()
```

- This function uses a parameter named person.
- A parameter is a variable that is initialized when the function is called.
- You have refactored/generalised the code

## Functions, Informally

- Our new main program:

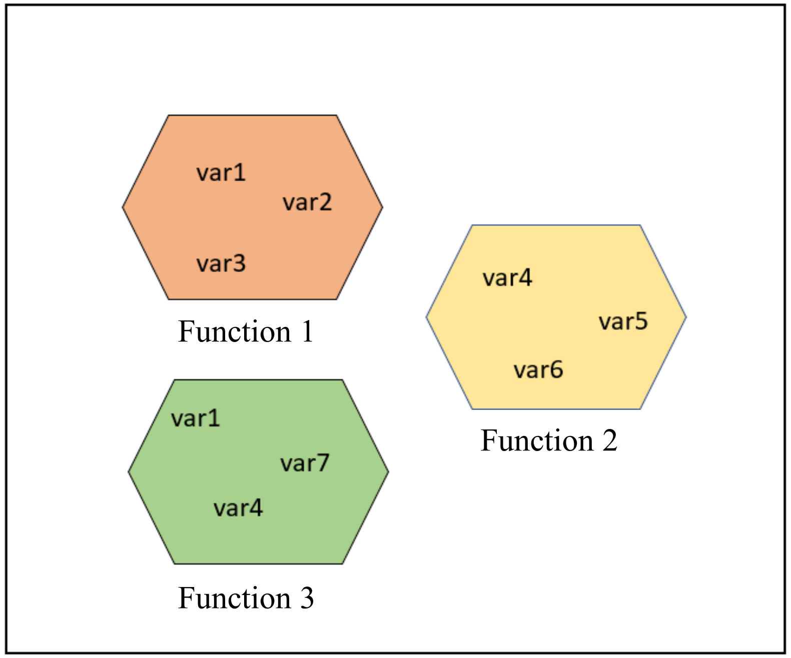
```
def main():
    sing("Fred")
    print()
    sing("Lucy")
```

- Gives us this output:

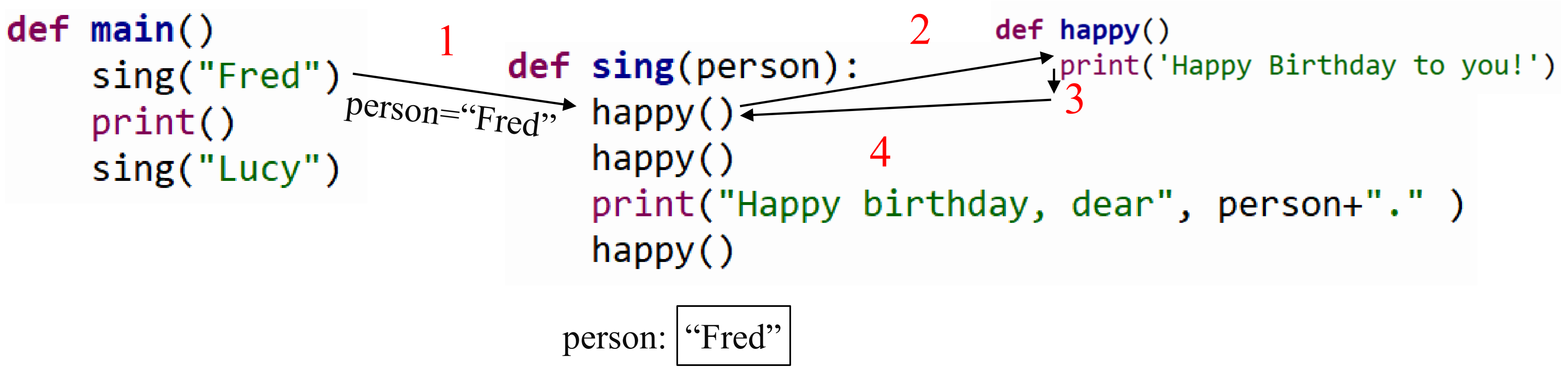
```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy.
Happy birthday to you!
```

## Scope of a Variable

- The *scope* of a variable refers to the places in a program a given variable can be referenced.
- The variables used inside of a function are *local* to that function, even if they happen to have the same name as the variables that appear inside of another function.
- The only way for a function to see a variable from another function is for that variable to be passed as a parameter (and it's the value that is passed!)



# Functions and Parameters: The Details



```
def main()
    sing("Fred")
    print()
    sing("Lucy")
```

```
def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person+"." )
    happy()
```

```
def happy()
    print('Happy Birthday to you!')
```

- When Python gets to the end of `sing`, control returns to `main` and continues immediately following the function call.
- The `person` variable in `sing` disappears when `sing` finishes!
- Local variables do **not** retain any values from one function execution to the next.

# Functions and Parameters: The Details

- A function is called by using its name followed by a list of actual parameters or arguments. <name>(<actual-parameters>)

  e.g., `sing("Fred")`

- When Python comes to a function call, it initiates a four-step process.

  1. *The calling program suspends execution at the point of the call.*
  2. *The formal parameters of the function get assigned the values supplied by the actual parameters in the call.*
  3. *The body of the function is executed.*
  4. *Control returns to the point just after where the function was called.*

## Functions and Parameters: The Details

- Let's trace through the following code:
  ```
  sing("Fred")
  print()
  sing("Lucy")
  ```
- When Python gets to `sing("Fred")`, execution of `main` is temporarily suspended.
- Python looks up the definition of `sing` and sees that it has one formal parameter, `person`.

## Passing Multiple Values: Future Value

- To find the future value of an investment, e.g., term deposit, we need three pieces of information.
  - *The principal sum*
  - *The interest rate*
  - *The number of years*

- These three values can be  supplied as parameters to the function.

## Passing Multiple Values: Future Value

- The resulting function looks like this:

```
def futureValue(p,n,r):
   print( p*(1+r)**n)
```

- To use this function, we supply the three values:

```
>>>futureValue(10000, 10, 0.1)
25937.424601…
>>>futureValue(10000, 20, 0.15)
163665.3739294…
```

## Functions and Parameters: The Details

- A function definition looks like this:

```
def <name>(<formal-parameters>):
    <body>
```

- The name of the function must be an identifier

- Formal-parameters is a list of variable names. The list can be empty, i.e., just `()`, if the function does not take any parameters.

# Multiple Function Parameters

- Functions can have multiple parameters.
- Formal parameters and actual parameters (arguments) are matched up based on *position*.
- As an example, consider the call to `futureValue`
- When control is passed to `futureValue`, these arguments are matched up to the formal parameters in the function heading:

```
def futureValue(p,n,r):
    print( p*(1+r)**n)


r = 0.1

>>>futureValue(10000, 10, r)

25937.424601…
```

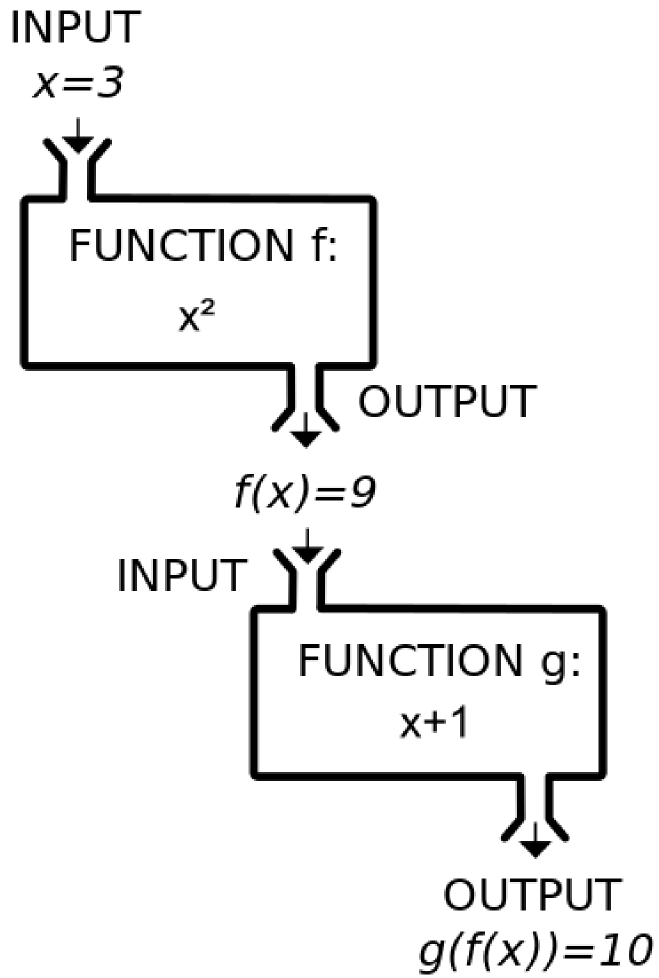
Value of r passed to the parameter r (which could have a different name!)

# Getting Results from a Function

- Passing arguments provides a mechanism for initializing the parameters declared in a function.
- Parameters are placeholders that represent the inputs a function expects to receive when it is called.
- Functions can also return values i.e., functions can have outputs

```
def f(x):
   return x*x
```

- When Python encounters `return`, it exits the function and returns control to the point where the function was called.
- In addition, the value(s) provided in the `return` statement are sent back to the caller as an expression result.



## Functions That Return Values

```
def f(x):
   return x*x
def g(x):
    return x+1

def main():
    x=3
    fx=f(x)
    gx=g(fx)
    print(“Output:”,gx)
```

## Returning Single Value: Future Value

- The resulting function looks like this:

```
def futureValue(p,n,r):
   return ( p*(1+r)**n)
```

- To use this function, we supply the three values:

```
>>>fv = futureValue(10000, 10, 0.1)
>>>print(fv)
25937.424601…
>>>futureValue(10000, 20, 0.15)
163665.3739294…
```

This value is not stored in any variable and is not available to be used later.

# Returning Multiple Values

- Simply list more than one expression in the return statement.

```
def sumDiff(x, y):
    sum = x + y
    diff = x - y
    return sum, diff
```

- When calling this function, use simultaneous assignment.

```
s, d = sumDiff(num1, num2)
```

- The values are assigned based on position, so `s` gets the first value returned (the sum), and `d` gets the second (the difference).

# All Functions Return a Value

- All Python functions return a value, whether they contain a `return` statement or not.
- Functions without a `return` hand back a special object, denoted `None`.
- A common mistake is writing a value-returning function but omitting the `return`!
  - *If your value-returning functions produce strange messages, check to make sure you remembered to include the* `return`*!*

## Summary

- We learned how to define new functions and how to call functions in Python.
- We learned how to write programs that use functions to reduce code duplication and increase program modularity.
- We learned how to pass multiple arguments to functions and how to return multiple values from functions.
- We studied the relationship between formal parameters actual parameters (arguments) and how functions can change parameters.