COMP 10280
Programming I (Conversion)

John Dunnion

School of Computer Science
University College Dublin

COMP 10280 Programming I (Conversion)/Lecture 15

Outline

Function Definition and Function Use

- There are two aspects to the use of our own functions in programming
- We first of all must define our function
- We then call or invoke (or use) the function wherever we want to use it

Defining and calling a function (1)

```
# Program to print out the largest of two numbers entered by the user
# Uses a function max

def max(a, b):
    """Function that returns the largest of its two arguments"""
    if a > b:
        return a
    else:
        return b



# Prompt the user for two numbers
number1 = float(input('Enter a number:  '))
number2 = float(input('Enter a number:  '))

biggest = max(number1, number2)

print('The largest of', number1, 'and', number2, 'is', biggest)

print('Finished!')
```

Defining and calling a function (2)

```
# Program to print out the largest of two numbers entered by the user
# Uses a function max
# Uses max in a print statement
def max(a, b):
    """Function that returns the largest of its two arguments"""
    if a > b:
        return a
    else:
        return b



# Prompt the user for two numbers
number1 = float(input('Enter a number:  '))
number2 = float(input('Enter a number:  '))

print('The largest of', number1, 'and', number2, 'is',
            max(number1, number2))

print('Finished!')
```

Defining and calling a function (3)

- The function, `max`, is introduced by the `def` keyword
- The function `max` returns a result via the `return` statement
- Since the function `max` explicitly returns a value, we can use the function anywhere we can use a value:
    - On the right-hand side of an assignment
    - As an item to be printed in a `print` statement
    - …

Defining and calling a function (4)

- The function `max` is defined to take two formal parameters
- When the function is called, we must supply two actual parameters or arguments
- In programming language design, there are a number of mechanisms for “passing” arguments between a function and the code that called it
- In Python, the mechanism used is a mixture of call by reference and call by value
- The mechanism used in Python is known as “Call by Object”, sometimes also called “Call by Object Reference” or “Call by Sharing”

Defining and using a Factorial function

```
# Program to calculate the facorial of a number
# Program prompts the user for an integer
# Uses a function to compute the factorial
# Question 1 from Practical Sheet 12

def fact(x):
    """Function that returns the factorial of its argument

    Assumes that its argument is a non-negative integer
    """
    res = 1
    for i in range(1, x+1):
        res *= i
    return res

# Prompt the user for an integer
number = int(input('Enter a number (an int >= 0):  '))

if number >= 0:
    print('The factorial of', number, 'is', fact(number))
else:
    print('Error:  can only calculate the factorial of a
            non-negative number (>= 0)')
```

Functions within functions

```
# Program to print out the largest of two numbers entered by the user
# Uses two functions:  max and print_max

def print_max():
    """Function that prints out the largest of two numbers

    Uses the function max to find the largest"""
    def max(a, b):
        """Function that returns the largest of its two arguments"""
        if a > b:
            return a
        else:
            return b

# Prompt the user for two numbers
    number1 = float(input('Enter a number:  '))
    number2 = float(input('Enter a number:  '))
    print('The largest of', number1, 'and', number2,
            'is', max(number1, number2))
    return


print_max()
```

Scoping (1)

Consider the following two programs:

```
# Program to print out the largest of two numbers entered by the user
# Uses a function max
# Uses max in a print statement
def max(a, b):
    """Function that returns the largest of its two arguments"""
    if a > b:
        return a
    else:
        return b



# Prompt the user for two numbers
number1 = float(input('Enter a number:  '))
number2 = float(input('Enter a number:  '))

print('The largest of', number1, 'and', number2,
            'is', max(number1, number2))

print('Finished!')
```

Scoping (2)

```
# Program to print out the largest of two numbers entered by the user
# Uses a function max
# Uses max in a print statement
def max(a, b):
    """Function that returns the largest of its two arguments"""
    if a > b:
        return a
    else:
        return b



# Prompt the user for two numbers
a = float(input('Enter a number:  '))
b = float(input('Enter a number:  '))

print('The largest of', a, 'and', b, 'is', max(a, b))

print('Finished!')
```

What is the difference in behaviour and output?

Scoping (3)

Consider the following example:

```
# Program to illustrate scoping in Python
def f(x):
    """Function that adds 1 to its argument and prints it out"""
    print('In function f:')
    x += 1
    y = 1
    print('x is', x)
    print('y is', y)
    print('z is', z)
    return x

x, y, z = 5, 10, 15

print('Before function f:')
print('x is', x)
print('y is', y)
print('z is', z)

z = f(x)

print('After function f:')
print('x is', x)
print('y is', y)
print('z is', z)
```

Scoping (4)

This program produces the following output:

```
Before function f:
x is 5
y is 10
z is 15

In function f:
x is 6
y is 1
z is 15

After function f:
x is 5
y is 10
z is 6
```

Stack Frames (1)

- In a Python program, the interpreter/run-time system uses a symbol table to keep track of all names defined and their current bindings
- When a function is called, a stack frame is created to keep track of all names defined in the function (including the formal parameters) and their current bindings
- The stack frame is "pushed" onto the stack of frames
- If there is a function called from within the function body, another stack frame is created for that invocation of the function and pushed onto the stack

Stack Frames (2)

- When the interpreter/run-time system tries to find an identifier (“resolve an identifier reference”), it checks the most local stack frame first, then the stack frames for enclosing functions and then the global stack frame (and then the built-in identifiers)
- When a function completes, its stack frame is disposed of (“popped” from the stack)
- The system never disposes of a stack frame from the middle of the stack
- It only ever removes the most recently added frame
- The strategy of stack frame disposal is this “last in, first out (LIFO)”
- Analogy: a stack of plates or trays in a cafeteria

Docstrings

- The text between the triple quotation marks at the beginning of a function is called a docstring in Python
- Python programmers use docstrings to provide specifications of functions
- One-line docstrings fit on a single line(!)
- Multi-line docstrings consist of a summary line just like a one-line docstring, followed by a blank line, followed by a more elaborate description
- The summary line may be used by automatic indexing tools; it is important that it fits on one line and is separated from the rest of the docstring by a blank line
- The docstrings can be accessed using the built-in function `help`
- Typing `help(function_name)` in the interpreter shell will return the docstring for the function `function_name`

Specifications

- A specification of a function describes the functionality (behaviour) of the function
- It defines a “contract” between the writer of a function and those who will use it
- This contract can be thought of as having two parts:
    - Assumptions: Conditions that must be met by users of the function
    - Guarantees: Conditions that must be met by the function, provided that it has been called with the assumptions having been satisfied
- “Pre-conditions” and “Post-conditions”

Decomposition and Abstraction

- Decomposition allows us to break down a problem into sub-problems
- In a programming context, we break down problems into modules
- These modules are reasonably self-contained and can be re-used in different settings
- Abstraction hides details
- It allows us to use a piece of code (function or entire program) as if it were a “black box”
- We can’t see, don’t need to see and shouldn’t want to see the internal details of the black box
- Abstraction should preserve information that is relevant in a particular context and forget information that is irrelevant

Recursion

- Recursion is a descriptive technique in which the description includes a reference to the thing being described
- While it appears often in programming, it is widely used as a descriptive technique

If you are...		then you are...
A	born in the island of Ireland to an Irish Citizen or to a non-Irish national who satisfied certain conditions at the time of your birth	an Irish citizen or entitled to Irish citizenship.
B	a child of **A**, born outside the island of Ireland	an Irish citizen.
C	a child of **B** and a grandchild of **A**, born outside the island of Ireland	entitled to Irish citizenship, but you must first register in the Foreign Births Register.
D	a child of **C** and a great-grandchild of **A**, born outside the island of Ireland	entitled to Irish citizenship, by having your birth registered in the Foreign Births Register, but *only* if your parent **C** had registered by the time of your birth.

Recursive definitions

- In general, a recursive definition comprises two parts:
    1. There is at least one base case that directly specifies the results for a special case
    2. There is at least one recursive case or inductive case that defines the thing in terms of the thing itself, typically a simpler version of it
- As the recursion proceeds, there is some notion of moving through increasingly simpler cases until we eventually reach one of the base cases

Recursive definition of the factorial function

- We have already seen the iterative definition of the factorial function:

$$n! = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ n \times n - 1 \times n - 2 \times \ldots \times 1 & n > 1 \end{cases}$$

- We note that the factorial of any number n, $n!$, is n times the factorial of the next smallest number, $n - 1$
- This leads to the following recursive or inductive definition:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n - 1)! & n \geq 1 \end{cases}$$

Recursive Python function to implement the factorial

```
def fact(x):
    """Function that returns the factorial of its argument

    Assumes that its argument is a non-negative integer
    Uses a recursive definition
    """
    if x == 0:
        return 1
    else:
        return x * fact(x - 1)
```