# Lecture 5 – Using Functions

M30299 Programming

School of Computing
University of Portsmouth

# The idea of functions

- We have already been using function definitions to allow us to write and test many small “programs” within a single file.
- Most real-world programs are much longer than those we have written so far.
- Typically, a program is a **collection** of several function definitions.
- The purpose of using functions is:
  - to help **break a large problem into smaller parts**;
  - to improve the **readability of code**; and
  - to **avoid repetition**—writing similar code over and over again.
- We will touch on each of these issues first, then see a slightly larger example at the end of the lecture; we’ll cover more realistic case studies later in the module.

# A program as a collection of functions

```
def say_hello():
    print("hello")

def say_goodbye():
    print("goodbye")

def main():
    say_hello()
    say_goodbye()

main()
```

# A program as a collection of functions

- This simple program just displays:

```
hello
goodbye
```

  on the screen.

- It consists of three function definitions, followed by a function call.
- The `main` function calls the other two functions (the name `main` is just a convention we'll use for the function that serves as a program **entry-point**).
- Executing our program will cause the `main()` function call at the bottom of the program to be executed, so the `main` function will be called.
- Program execution finishes after this function call is completed.

# Using functions to break down large problems

- Clearly, to write a program to carry out such a simple task we don't really need to use several functions.
- However, let's briefly consider a more complicated problem:

  Write a program that reads data about employees (salary, overtime hours, etc.) from a file, and then displays how much each should be paid this month.

- A first step in **designing** a solution to this problem might be to break the problem down into three simpler **sub-problems**:
  - read employees' data;
  - calculate wages; and
  - display wages.

# Using functions to break down large problems

- Each could be solved using a function, and `main` would be:

```
def main():
    ... readEmployees(...)
    ... calculateWages(...)
    ... displayWages(...)
```

- It is possible that the sub-problems could be broken down further. E.g., to calculate the wages of an employee involves:
  - calculating basic pay;
  - calculating overtime pay; and
  - deducting tax.
- The solution to these sub-problems might be functions themselves.

# Functions with parameters

- Imagine that we want to display greetings to several different people. We might write the following:

```
def main():
    print("Hello, Vicky. How are you today?")
    print("Hello, Tom. How are you today?")
    print("Hello, Fred. How are you today?")
    print("Hello, Sam. How are you today?")
    print("Hello, Gemma. How are you today?")
```

- This would work, but it contains a lot of repeated code: the only difference between the greetings is the person's name.
- We define a function that has the name as a **parameter**...

# Functions with parameters

```
def greet(name):
    print("Hello " + name + ".", end=" ")
    print("How are you today?")
```

- The parameter `name` is a special variable whose value is initialised when the function is **invoked** or **called**.
- When we call this function we supply an **argument** (a value for the parameter):

```
>>> greet("Sam")
Hello Sam. How are you today?
>>> greet("Fred")
Hello Fred. How are you today?
```

# Functions with parameters

- Our `main` function can now be replaced by:

```
def main():
    greet("Vicky")
    greet("Tom")
    greet("Fred")
    greet("Sam")
    greet("Gemma")
```

- Apart from the overall reduction in text, what other advantage(s) might this new code – `greet` & `main` – have?

# Functions that return values

- We have used built-in functions that **return** values to the caller:

```
>>> euros = float(input("Enter amount in euros: "))
>>> print(math.sqrt(2))
1.4142135623730951
```

- Functions `float`, `input` and `math.sqrt` all return values (`print` doesn't).
- Notice that the **calls** to these functions are **expressions** (they have **values** and often appear on the right-hand-side of assignments).
- Let's write our own function that returns values:

```
>>> def square(x):
        return x * x
```

# Functions that return values

- A **return statement** like this causes:
  - the function to exit (control is passed back to the caller), and:
  - the returned value is given as the function call's value.
- Let's use our user-defined `square` function:

```
>>> print(square(2))
4
>>> z = 3
>>> y = square(z)
>>> print(y)
9
```

# Functions that return multiple values

- We note that functions can return more than one value; e.g:

```
>>> def sum_and_difference(n1, n2):
        return n1 + n2, n1 - n2
```

returns both the sum and the difference of two numbers.

- Such functions are often used together with a **simultaneous assignment**:

# Functions that return multiple values

- We can thus take the two values returned by

```
>>> def sum_and_difference(n1, n2):
        return n1 + n2, n1 - n2
```

and assign them to two separate variables:

```
>>> s, d = sum_and_difference(10, 3)
>>> s
13
>>> d
7
```

# Functions cannot change argument values

- Consider the following code

```
def turn_up_heat(temp):
    temp = temp + 10

def main():
    temperature = 15
    turn_up_heat(temperature)
    print(temperature)
```

- Calling `main` will result in the output value 15.
- This is because only the (local) parameter `temp`'s value is changed inside the `turn_up_heat` function; the value of `temperature` is not changed.

# Functions cannot change argument values

- To change `temperature`'s value, we can rewrite our code to use a function that **returns** a new value, and then assign this value to `temperature` in `main`.

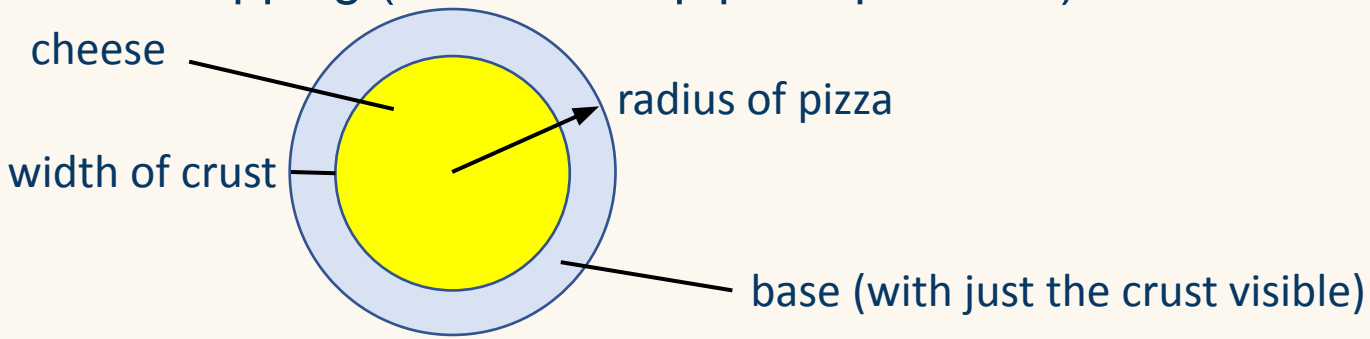
```
def hotter_temp(temp):
    return temp + 10

def main():
    temperature = 15
    temperature = hotter_temp(temperature)
    print(temperature)
```

- We see that:
  - the call `turn_up_heat(temperature)` is a **statement**; and
  - the call `hotter_temp(temperature)` is an **expression**.

# Writing function definitions – an example

- Suppose that the cost of a cheese pizza is made up of:
  - the cost of the dough base (which is 1p per square cm);
  - the cost of the cheese topping (which is 2.5p per square cm).




- Let's write program that asks the user for:
  - the radius of the pizza (i.e. the radius of the base); and
  - the width of the crust;

  and displays the total cost of the pizza in pounds.

# Splitting the task into two functions

- We'll start by splitting the task into two separate functions:
  - `main` - this will ask the user for the two inputs, and displays the final result (i.e. it forms a **user interface**.)
  - `cost_of_pizza` - this function will actually calculate the cost.
- Clearly, the `main` function will need to call `cost_of_pizza`; it will take the form:

```
get radius of pizza from user
get width of crust from user
call cost_of_pizza to calculate cost of pizza
      based on the radius and crust width
display cost
```

# The `cost_of_pizza` function

- The `cost_of_pizza` function needs the pizza **radius** and crust **width** in order to calculate the cost: these will be its **parameters**.
- To give the cost the function can follow the steps:
  - cost of base = 0.01 × area of base
  - cost of topping = 0.025 × area of topping
  - return cost of base + cost of topping
- We could therefore write:

```
def cost_of_pizza(radius, width):
    cost_base = 0.01 * math.pi * radius ** 2
    cost_topping = 0.025 * math.pi * (radius - width) ** 2
    return cost_base + cost_topping
```

# The `cost_of_pizza` function

- However, there is repetition here (in the calculation of areas).
- A better solution would be to use an extra function that will find areas of circles:

```
def area(radius):
    return math.pi * radius ** 2

def cost_of_pizza(radius, width):
    cost_base = 0.01 * area(radius)
    cost_topping = 0.025 * area(radius - width)
    return cost_base + cost_topping
```

# The `main` function

- The `main` function is relatively simple to write, and we add a call to `main` at the bottom to complete our program:

```
def main():
    base = int(input("Pizza radius: "))
    crust = int(input("Crust width: "))
    cost = cost_of_pizza(base, crust)
    print(f"The cost is {cost:.2f} pounds.")

main()
```