

CP1401 Module 6

Functions 1



Learning outcomes – you will be able to:

- Implement Python programs using functions
- Understand input parameters and returning data
- Understand and use the Single Responsibility Principle (SRP)
- Name functions well using meaningful verb phrases

(Later, in Functions 2)

- Test functions with meaningful test values
- Design larger programs using smaller functions (decomposition)





Do this now

- Write a program to ask the user for their age, making sure it's valid (0-150); then print whether they are:
 - child (0-17)
 - adult (18-64)
 - geriatric (65+)
- What data should you use to test this program?



Pseudocode

```
get age
while age < 0 or age > 150
    print error message
    get age
if age < 18
    print child
else if age < 65
    print adult
else
    print geriatric
```

Standard while loop pattern

if-elif-else pattern



Python

```
age = int(input("Age: "))
while age < 0 or age > 150:
    print("Invalid age")
    age = int(input("Age: "))
if age < 18:
    print(f"At {age} years old, you are: child")
elif age < 65:
    print(f"At {age} years old, you are: adult")
else:
    print(f"At {age} years old, you are: geriatric")
```



This next bit is so random

- Remember our guessing game:

```
SECRET = 6
guess = int(input("? "))
while guess != SECRET:
    print("Guess again!")
    guess = int(input("? "))
print("You got it!")
```

- How can we make this more interesting by making the secret **random**?
 - By using the random module



Use the random module to do random things

```
import random
```

```
secret = random.randint(1, 10)  
print(secret)
```

- First, we need to **import** the random **module**.
- Then we have access to many new functions, including `randint`.
- `randint` takes 2 parameters, the low and high bounds for generating a random number (1-10 inclusive in this example).



random.randint can use any meaningful arguments

```
import random
```

```
low = int(input("Low: "))  
high = int(input("High: "))  
secret = random.randint(low, high)  
guess = int(input("? "))  
while guess != secret:  
    print("Incorrect")  
    guess = int(input("? "))  
print("Got it!")
```

Notice that this is now **secret**
instead of **SECRET**.
Never name variables like
CONSTANTS



So, what have we got here? Let's break it down.

```
secret = random.randint(low, high)
```

- random is a **module** - a file that contains other code we can use
- randint is a **function** - a reusable piece of code that we can **call**
- low and high are **arguments** - the parameters that change how the randint function works when we call it
- secret is a variable that stores the what the randint function **returns**

The random module contains many more functions, but that will do us for now.





Do this now

Write a complete program (including import at the top) that asks the user for their age, then tells them how old they will be in a random number of years (where that random number is between 1 and their age)

Example:

Age: 42

In 40 years, you will be 82





As programs get larger, we need constructs to manage them

- Larger programs can sometimes be written all in one section, but:
 - Large programs can be hard to read –
Programmers sort of have to hold the whole thing in their mind at once instead of thinking of just one piece at a time
 - We may have **code that performs the same task in multiple places**
 - Repetition structures can help with reducing duplicated code, but not always
 - Often, our repeated tasks are really like a sub-program, or... **function!**



Functions allow us to break larger programs into smaller, more manageable pieces

- A function is a named group of statements within a program that performs a specific task
 - Usually one task of a large program
- We can break off chunks and solve them as separate parts
 - This is **decomposition** - decomposing a large problem into smaller problems that are easier to solve
 - When the problems are smaller versions of the same overall problem, this is known as the *divide and conquer* approach



Here's a very simple function example

```
def print_line():  
    print("-" * 20)
```

```
print_line() # -----
```



Let's add a parameter to customise the function

```
def print_line(length):  
    print("-" * length)
```

```
print_line(20)    # -----
```



Here's a complete program using that function

```
def main():  
    print_line(20)  
    print("Welcome!")  
    print_line(8)
```

main function **definition**

print_line function **call**

```
def print_line(length):  
    print("-" * length)
```

print_line function **definition**



main() ← main function **call**

Welcome!

Deconstructing function **definitions** in Python

```
def function_name(parameters):
```

 statement
statement

- First line includes the keyword `def` followed by a function name, then brackets with a list of parameters followed by a colon.

This is the function **header**.

- We can have zero, one, or more parameters
- Indenting defines the **body** of the function



Deconstructing function **calls** in Python

```
function_name(arguments)
```

- We've done this a lot already:
 - `print`, `input`... what other built-in functions have we used?
- Simply use the function's name followed by round brackets that contain any arguments. `print("Hello world")`
- If a value is returned, then we probably want to use it:
`age = int(input("Age: "))`
`age = get_age()`



We can pass arguments to functions that take them

- An argument is a piece of data that is sent into a function
 - In `print("Hello")`, "Hello" is the argument.
 - Functions can use arguments in their bodies, so we use them to customise what a function does (like we did with random numbers)
 - An argument can be anything that evaluates to a value (expression), including a variable, constant, other function, a literal... Example:

```
name = input("Name: ")  
print("Hello", name)
```



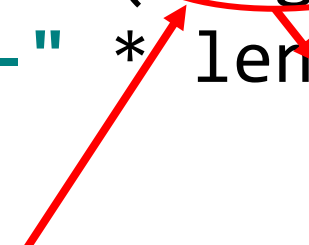
What are all the arguments here?

Arguments get passed to function parameters

- "Parameter" is what we call the variable inside the function that is assigned the value of an argument when the function is called
 - A parameter's **scope** is the whole function in which the parameter is used.

```
def print_line(length): Parameter  
    print("-" * length) length = 10
```

```
print_line(10) Argument
```

A red arrow points from the argument '10' in the function call to the parameter 'length' in the function definition, illustrating the passing of the argument's value to the parameter.

We can pass multiple arguments (if the function has multiple parameters)

- Python allows writing functions that accept multiple arguments
 - The parameters passed in are referred to as a parameter list
 - Parameter list items are separated by comma(s)
- Arguments are passed by position to corresponding parameters
 - The first parameter receives value of first argument, second parameter receives value of second argument, etc.

```
def print_grid(rows, columns):  
    # do this now (print a grid of *s)
```

```
print_grid(3, 7)
```

These are called "positional arguments"





Do this now

```
def print_grid(rows, columns):  
    # complete this function to print a grid of *s
```

```
print_grid(3, 7)
```

```
*****
```

```
*****
```

```
*****
```



Many functions return values

- We often write code like:

```
name = input("Name: ")  
number = int('42')  
value = random.randint(1, 10)
```

- So, these functions all **return** a value
 - This means these function calls **evaluate** to the value they **return**
 - `int('42')` evaluates to the integer 42
- We can write our own functions that return values



Use the **return** statement to return a value

- Consider a function to calculate the BMI (Body Mass Index), as seen in practicals

```
def calculate_bmi(height, weight):  
    return weight / (height ** 2)
```

- The function has two parameters, height and weight. These could come from user input, random numbers or whatever.
 - It does not ask the user for these inputs
- The function **returns** the result of the calculation
 - It does not print the value



Here's the complete BMI program

```
def main():  
    height = float(input("Height (m) : "))  
    weight = float(input("Weight (kg): "))  
    bmi = calculate_bmi(height, weight)  
    print(f"Your BMI is {bmi}.")
```

```
def calculate_bmi(height, weight):  
    return weight / (height ** 2)
```

Here's another program that uses the same function

```
def main():  
    height = random.uniform(1, 2)  
    weight = random.randint(44, 99)  
    if calculate_bmi(height, weight) < 15:  
        print("Not considered overweight")
```

```
def calculate_bmi(height, weight):  
    return weight / (height ** 2)
```



main()

See how we can use the exact same function for a different program with no user inputs and not printing the result?

Functions should "do one thing"

- The **Single Responsibility Principal** (SRP) is the most important function design consideration
- **Functions should not usually print** unless that's their one thing
 - Instead, they should **return their result**
 - Then we can use the returned value to print or do other things
- Functions should not usually get user input, unless that's their job
 - Then we can get the required inputs from user or other places
- If a function does too many things, it should probably be "refactored" into multiple functions



These functions each have a single responsibility

- This function's one job is to get a valid age. We can customise it by passing in low and high parameters.
 - Notice it does not print; it returns.
- This function's one job is to print a grid of asterisks. We can customise it by passing in rows and columns parameters.
 - Notice it does not get user input; it takes arguments

```
def get_valid_age(low, high):  
    age = int(input("Age: "))  
    while age < low or age > high:  
        print("Invalid age")  
        age = int(input("Age: "))  
    return age
```

```
def print_grid(rows, columns):  
    for i in range(rows):  
        for j in range(columns):  
            print("*", end="")  
        print()
```



Take a coffee break to learn more about functions :)



Functions should be reusable

- Making coffee involves many small steps, which are grouped into overall tasks. Our functions for making coffee are:
 - grind_beans()
 - pour_espresso()
 - steam_milk()
 - combine()
- We can reuse these functions in different ways:
 - Flat white = grind_beans() + pour_espresso() + steam_milk() + combine()
 - Espresso = grind_beans() + pour_espresso()
 - Hot chocolate = make_chocolate() + steam_milk() + combine()
 - Take-home grinds = grind_beans()



Don't limit function reusability

- What if we tried to make it simpler by making the first step automatically call the second step and so on?
 - `grind_beans()` calls `pour_espresso()` calls `steam_milk()` calls `combine()`
- Seems easier, but...
what then if we wanted an espresso or a hot chocolate?
 - NO.
When we try to get just an espresso, it automatically goes on to steam milk...
When we just want steamed milk for a babycino or hot chocolate...
Also not possible.



Functions are often used for sections of a program

- Let's look at the age program we wrote earlier and think about the **sections** in it.



Python

```
age = int(input("Age: "))
while age < 0 or age > 150:
    print("Invalid age")
    age = int(input("Age: "))
if age < 18:
    print(f"At {age} years old, you are: child")
elif age < 65:
    print(f"At {age} years old, you are: adult")
else:
    print(f"At {age} years old, you are: geriatric")
```

Input "section"

Another section



Let's break that up into 2 sections using functions:

```
def main():  
    age = get_age()  
    if age < 18:  
        print(f"At {age} years old, you are: child")  
    elif age < 65:  
        print(f"At {age} years old, you are: adult")  
    else:  
        print(f"At {age} years old, you are: geriatric")
```

```
def get_age():  
    age = int(input("Age: "))  
    while age < 0 or age > 150:  
        print("Invalid age")  
        age = int(input("Age: "))  
    return age
```



Now we've got a main program in two sections

```
def main():  
    age = get_age()  
    if age < 18:  
        print(f"At {age} years old, you are: child")  
    elif age < 66:  
        print(f"At {age} years old, you are: adult")  
    else:  
        print(f"At {age} years old, you are: geriatric")
```

```
def get_age():  
    age = int(input("Age: "))  
    while age < 0 or age > 150:  
        print("Invalid age")  
        age = int(input("Age: "))  
    return age
```

main()

main() looks like the **whole** program - we can read it and understand it all (not all details)
Here, we **call** get_age()

get_age() is a function that represents a section of the main program

Last, we **call** main()



Python programs should follow this structure

```
"""module-level docstring"""  
import statements  
CONSTANTS
```

```
def main():  
    statements
```

```
def do_step_1():  
    statements
```

```
def do_step_2():  
    statements
```

```
main()
```

- top docstring for documentation
- all imports next
- define any CONSTANTS next
- then main is *always* the first function
- then define any/all other functions
- lastly, *call* main to run the program



main should "look like" the whole program

- You should be able to read (only) main and understand the whole program, with the details "abstracted away" in other functions.
- Functions should *not* call other functions that come next, but functions *can* call functions that are sub-steps of the same job

Bad:

```
def main():  
    do_step_1()  
    def do_step_1():  
        ...  
        do_step_2()
```

Reading this bad main, it looks like the program is just step1. Also (even worse), you can not run step1 without also running step2.

Good:

```
def main():  
    do_step_1()  
    do_step_2()  
  
def do_step_1():  
    do_step_1_part_1()  
    do_step_1_part_2()
```





Do this now

Write a **complete** Python program following the standard structure that uses a main and a function similar to the print_line one we saw earlier, but that prints a line of random length up to the parameter passed in

E.g., print_random_line(5) would print a line of length 1-5.

Ask the user for a number then print a line up to that long.

Version 2: also ask for a number of lines to print and use a loop to print that many lines.

```
"""module-level docstring"""  
import statements  
CONSTANTS
```

```
def main():  
    statements
```

```
def do_step_1():  
    statements
```

```
def do_step_2():  
    statements
```

```
main()
```



Name functions using verb phrases

```
age = get_age()
```

- age sounds like a *noun*, so it's a thing, so it's a *variable*
- get_age sounds like a *verb*, so it's a *function* that does something
- Good function names usually complete the sentence:
"*This function will...*"
 - "This function will get_age" 😊
 - "This function will result_calc" 😞
 - "This function will determine_grade" 😊
 - "This function will value_check" 😞



Name functions using verb phrases

- A function's name should *say what it will do*
- Functions that return Booleans (True or False) are usually used as conditions, and should usually be named like **is_***

Purpose	Bad Names	Good Name
Determine if a temperature is hot	hotness, how_hot, calculate_hot, hot...	is_hot
Display a report	report, calculate_report, results...	display_report
Get a user's name	name_getter, namey_mcname_face...	get_name





Do this now

- Write good names for functions for the following tasks:
 1. determine a subject grade based on a given total score
 2. convert currency from USD to AUD
 3. print a report
 4. calculate the average of a list of numbers
 5. determine if a number is even
 6. get a user's salary, making sure that it is not negative



Functions can be used to "abstract" a task

- We very often have to write code that performs the same kind of task multiple times
- We already know there are disadvantages to duplicating code
 - Makes program large; can be time consuming; may need to make changes in many places...
- Some functions represent a task we want to perform multiple times but with slight differences
 - Calculating the area of a circle (of a given/different radius)
 - Converting values in one unit of measurement to another
 - Generating a random number between two values

```
5 Get 5 numbers and display their total and average

total = 0
get number1
get number2
get number3
get number4
get number5
total = number1 + number2 + number3 + number4 + number5
average = total / 5
display total and average
```

This should make alarm bells ring!
What if there were 5000 numbers?!



Let's reconsider the age program from before

```
def main():  
    age = get_age()  
    if age < 18:  
        print(f"At {age} years old, you are: child")  
    elif age < 65:  
        print(f"At {age} years old, you are: adult")  
    else:  
        print(f"At {age} years old, you are: geriatric")
```

Is there another "task" here that we could use a function for?

```
def get_age():  
    age = int(input("Age: "))  
    while age < 0 or age > 150:  
        print("Invalid age")  
        age = int(input("Age: "))  
    return age
```



Many functions do some kind of "processing"

- Remember where we started, with Input, Processing, Output?
- We might like to turn each step into a function (if it makes sense)
- Here, the processing step is determining an age category based on a person's age

```
if age < 18:  
    print(f"At {age} years old, you are: child")  
elif age < 65:  
    print(f"At {age} years old, you are: adult")  
else:  
    print(f"At {age} years old, you are: geriatric")
```



But... the processing and output steps are combined

- We need to separate these two steps so that the processing step is a reusable function (**return** not print)

```
def determine_category(age):  
    if age < 18:  
        return "child"  
    elif age < 65:  
        return "adult"  
    else:  
        return "geriatric"
```



Now we have...

```
def main():
    age = get_age()
    category = determine_category(age)
    print(f"At {age} years old, you are: {category}")

def get_age():
    age = int(input("Age: "))
    while age < 0 or age > 150:
        print("Invalid age")
        age = int(input("Age: "))
    return age

def determine_category(age):
    if age < 18:
        return "child"
    elif age < 65:
        return "adult"
    else:
        return "geriatric"
```



What have we gained?

- We can focus our development efforts on one function at a time
- We can reuse these functions (in this or other programs)
 - `get_age` could be used anywhere we want a valid age
 - `determine_category` could be used for a ticket sales program with different prices per age category...
- We can read/understand the "whole program" (main) very quickly
 - We don't know all the details but we can see what it does overall

```
age = get_age()  
category = determine_category(age)  
print(f"At {age} years old, you are: {category}")
```



There are benefits to using functions

- Code can be reused
 - write the code (function) once and call it multiple times
- Testing and debugging is easier
 - Can test and debug each function individually
- Teamwork is easier
 - Different team members can write different functions
- Code is more readable (easier to understand)
 - Good names mean you can read the function header and know what it does
- Large programs can be developed faster





Do this now

1. Write a function that prints a string a certain number of times, where the string and the number of times are both input parameters (arguments)
 - Call it to print your first name 4 times
 - Call it to print "Hi" 17 times

2. Write a function to calculate the average of two numbers
 - Call it to work out the average of 7 and a user's age



Now do these next steps

- (Like we did with making coffee)
Find an everyday process and think about breaking it down into functions - things that are sections or reusable tasks
- Practise writing programs that use functions
 - Go back to some of your earlier pracs and rewrite them with meaningful and well-named functions

