CP1401 Module 7
Functions 2

Learning outcomes – you will be able to:

- Understand scope and lifetime of variables
- Test functions with meaningful test values
- Design larger programs using smaller functions (decomposition)
- Use module and function docstring comments

(Earlier, in Functions 1)

- Write Python code to implement programs using functions
- Understand input parameters and returning data

Do this now

- Rewrite the Happy Products program from prac 5 using functions
  - main (menu)
  - calculate (section)
  - print instructions
  - maybe more… like getting a valid number

- If you are keen, use PyCharm's **Refactor > Extract Method** tool!

Do not use global variables

- Global variables are variables defined above/outside a function.
- Global variables make debugging difficult.
  - Many locations in the code could change the value.
- Functions that use global variables are usually dependent on those variables, which makes it hard to transfer to another program
- Global variables make a program harder to understand.
- So… **do not define variables outside functions**.
  - (If you have no functions, *everything* is global, so this is fine.)

Do use global constants

- Global constants are variables named in ALL_CAPS that cannot (will not) be changed, defined above/outside a function.
- It's permissible to use global constants in a program. Why?
  - Because no functions will change them (they're constant)
- As per our standard structure, declare constants at the top

```
PI = 3.14159265358979323
```

Follow the naming conventions

- Global constants are variables named in ALL_CAPS
- So, if you ever find yourself typing something like:

```
PI = PI + 1
```

or

```
NUMBER_OF_NIGHTS -= 1
```

Then you KNOW you're doing something wrong, because CONSTANTS should NEVER be changed after they are set.

- Either you should not be changing the constant or the name should be just like a variable (`not_all_caps`)

Local variables are defined inside functions

- A local variable belongs to the function in which it was created
  - Only statements inside that function can access it
  - An error will occur if another function tries to access the variable
- **Scope**: the part of a program in which a variable may be accessed
  - For local variable: function in which created
  - Functions can not see other functions' local variables
- A variable's **lifetime** is the time that it is "alive", whether it is accessible or not
  - Similar to, but not the same as, scope

Scope and lifetime

At this point, two different variables called **x** exist (lifetime), but only the local variable (equal to 3) is in scope

```
def demo():
    x = 3
    y = 4
    print(x, y)  # prints 3 4
```

Execution starts here

At this point, the function variables called **x** and **y** cease to exist (their lifetime is over)

```
x = 7
print(x) # prints 7, not 3
demo()
print(y) # ERROR! y doesn't exist
```

Tracing how functions execute

```
def main()
    statement a
    function_one()
    statement b

def function_one()
    statements
    function_two()

main()
```

- When a function is called:
  - Interpreter jumps to the function and executes statements in the block
    - the original function is not finished, but it's suspended, waiting for execution to return
  - When that function is finished, interpreter jumps back to part of program that called the function and goes to the next line

IT@JCU

Now that we know how functions work, how can we plan to use them?

JAMES COOK UNIVERSITY AUSTRALIA

Look for functions during design

- Functions are a design tool for grouping related code
- Unlike basic constructs (decision, repetition, processing), you won't be looking for specific or implied words in a problem description
- When you start planning your algorithm, you'll see that you want to group code together with a name - to create a function

Top-Down Design

- Top-down design is a useful technique for breaking an algorithm into functions
- The main task is broken into smaller sub-tasks
- Each of the sub-tasks could be broken into other smaller sub-tasks
- Each sub-task should do "one" thing (Single Responsibility)
  - implemented as a function - with a good name
- Reading through just the function names in context should explain (mostly) what the program does
  - If a function name says it does two things, it probably should be two functions

Problem Description

Ask user for their name (must not be empty),
then ask for each of their subject names,
then each of their assessment scores per subject.
Add up the total of those scores (out of 100),
then tell them what grade they will receive for each subject.

Pseudocode for functions

Function definitions

```
function do_something(x, y)
    return (x + y) * (y - x)
```

Function calls

```
result = do_something(azimuth, altitude)
```

Do this now

- Write the pseudocode for JUST the main function for the following
  - That is, you need to identify the functions but not define them

- Jerry's car's speedo shows miles (mph) instead of kilometres per hour (kph). He wants to be able to enter his speed in mph, the speed limit in kph and determine if he will get any speeding fine.

  - Out of interest, remember:
    prac 2 where we calculated km -> m and
    prac 3 where we determined speeding fines

Jerry's not doing this while driving, is he?

- Write the pseudocode for JUST the main function for:
- Jerry wants to be able to enter his speed in mph, the speed limit in kph and determine if he will get any speeding fine.

```
function main()
    speed_in_mph = get_valid_number("speed in mph")
    speed_in_kph = convert_miles_to_km(speed_in_mph)
    speed_limit_in_kph = get_valid_number("speed limit in kph")
    fine = determine_fine(speed_in_kph, speed_limit_in_kph)
    print fine
```

Jerry's not doing this while driving, is he?

- Write the pseudocode for JUST the main function for:
- Jerry wants to be able to enter his speed in mph, the speed limit in kph and determine if he will get any speeding fine.

```
function main()
    speed_in_mph = get_valid_number("speed in mph")
    speed_in_kph = convert_miles_to_km(speed_in_mph)
    speed_limit_in_kph = get_valid_number("speed limit in kph")
    fine = determine_fine(speed_in_kph, speed_limit_in_kph)
    print fine
```

Do this now

- Write pseudocode for the worker level question from one of our earlier practicals - with functions (the whole program):

```
worker_level = int(input("Worker level: "))
while worker_level < 1 or worker_level > 10:
    print("Invalid worker level")
    worker_level = int(input("Worker level: "))
salary = 5000.0 * worker_level
print("Level {worker_level}, Salary ${salary:,.2f}")
```

IT@JCU

Docstrings are expected for good code

Add a docstring to
every function you write

JAMES COOK UNIVERSITY AUSTRALIA

Python has two kinds of comments

- Block / inline comments
  - Start with # ("hash", not "hash tag") then a space
  - One-line, short

```
# This is a comment
```

- Docstrings
  - Start and end with triple quotes """
  - Only used for modules, functions, classes, not in between normal code

```
"""
CP1401 - Coding Checkpoint 1
https://github.com/CP1401/Practicals
"""
```

Don't put "noise" in your comments

- Avoid using more words than you need in order to make your comment clear because using more words just means that the reader of your code has more words to read so they spend more time just reading things that they really already know because the context makes it clear, like, you know if there's a function docstring and you're writing a docstring for this function and the reader is reading this docstring that we're talking about here and they already know it's, like, a docstring, so you know as they read it, if it says something like "this function will"… they already knew it was a function docstring so the only possibility for this comment is that because it's a function docstring it must therefore, by definition, without question be 100% definitely explaining what the function does. So… don't waste words.

Function docstrings must be the *first* line inside a function definition

```python
# This is NOT a docstring.
"""This is NOT a docstring."""
def do_something():
    """This IS a docstring."""
    print()
    """This is NOT a docstring"""
```

Write good docstrings

- PEP257 describes the standards for writing good docstrings. Follow these guidelines. https://www.python.org/dev/peps/pep-0257/
- One-line docstrings are enough for now.
  Start with a capital, end with a full stop.
  Don't put spaces just inside the quotes.
  No "noise"; e.g. *"This function..."* is unnecessary.
- Good: `"""Determine largest value in list."""`
- Bad: `""" this function will find the largest value in the list that is passed in as a parameter and return it """`

IT@JCU

Function docstrings say what the function **will** do

- All comments (not just docstrings), should be written in the *imperative* mood, e.g.,
  `"""Convert a to b."""` (not "*converts*")
- Notice how this example completes the sentence, "this function will…"

```
def is_valid_password(password):
    """Determine if password is considered valid."""
    if len(password) < 6 or " " in password:
        return False
    return True
```

Docstrings can help clarify details

- Notice how this example helps us know what units of measurement the function expects

```
def calculate_bmi(height, weight):
    """Calculate BMI using height in metres and weight 
in kilograms."""
    return weight / (height ** 2)
```

(Please ignore the word-wrapping to fit in a PowerPoint slide)

Some docstrings sound just like function names...

- … and that's just fine

```python
def print_line(length):
    """Print a line of length hyphens."""
    print("-" * length)
```

Don't use the wrong tool for the job

E.g., Recursion (function calls itself) is **only** for recursive problems, not for looping

IT@JCU

Functions should be testable
How could we test these two similar functions?

```python
def age_category():
    age = int(input("Age: "))
    if age < 18:
        category = "child"
    elif age < 65:
        category = "adult"
    else:
        category = "geriatric"
    print(category)
```

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

Functions should be testable
How could we test these two similar functions 100x?

```
def age_category():
    age = int(input("Age: "))
    …
    print(category)

age_category()
…
```

Oh no… ☹

```
def determine_category(age):
    …
    return "geriatric"

for age in range(101):
    category = determine_category(age)
    print(f"{age} is {category}")
```

Nice ☺

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 algorithms and programs that use functions
  - Go back to some of your earlier pracs and rewrite them with meaningful and well-named functions