Functions
What is a function?

A function is a named sequence of statements that performs a computation.

When we define a function, we write the name and the sequence of statements.

To use the function, we call the function by name.

Functions: properties

- Functions can be defined as a block of re-usable code to perform specific tasks.
- Reusable code blocks only need to be written once, then they can be used multiple times.
- Functions can be part of the language (built-in functions) or can defined by the users (user-defined functions).

Functions are called inside the codes with:

1. input parameters (arguments)
2. return values
Functions: example

We have already used some function calls:

\[ \text{e.g.: } \text{type}(3.14) \]

The name of the function is:
\[ \text{type} \]

The expression in parentheses is the argument of the function:
\[ (3.14) \]

The result, for this function, is the type of the argument:
\[ \text{<class 'float'>} \]

Why do we use functions?

- **Modularity**: functions break a program into smaller and modular blocks.
- **Organization**: functions can help manage and organize larger programs
- **Simplification**: functions avoid repetition and makes code reusable.
- **Reusability**: Once a function is defined, it can be reused several times. We can invoke the same function many times at different location in a program.
- **Abstraction**: Functions provide a high-level view of the program components whose detailed can be filled in later.
Functions: definition

Functions encapsulate a task. Here’s how they are defined in Python.

```python
def function_name (formal_parameters):
    '''Optional comment, which is called a docstring, describing your function.'''
    <function body>
    return <some value or values> # optional
```

Functions: docstring

The docstring (documentation string) is a string placed after the header. It describes the function.

It is a good programming practice to include a docstring in a function for documentation

```python
>>> def check_parity(number):
...     """This function returns the parity of a number""
...     parity = "even" if number % 2 == 0 else "odd"
...     return parity
...     print (check_parity(6))
...     print (check_parity(21))
...
  even
  odd
```
Functions: call

You call the function as follows:

```
function_name (actual_parameters)
```

This has the effect of executing the function body with the formal parameters replaced by the actual parameters.

Functions: example

What is the execution order and output of the following program?

```python
1 def print_msg():
2     print('I love Python!')
3 
def is_even(num, divisor):
4     print(num % divisor == 0)
5 
6 print_msg()
7 is_even(10, 2)
8 is_even(7, 10)
```

<table>
<thead>
<tr>
<th>Execution order</th>
<th>Output</th>
</tr>
</thead>
</table>
| 1, 4, 7, 1, 2, 8, 4, 5, 9, 4, 5 | I love Python!  
True  
False |
Functions: main()

- The `main()` function is a special function that runs automatically when the program is executed.
- It is not required to use `main()` in Python, but it is a good programming practice for the logical structure of the code.
- There is nothing special about the name `main()`.
- We can rename the function `main()` if used as a regular function.
- The name `main()` is consistent with some of the other programming languages.

In C, C++, C#, Java et., all these programming languages require the `main function` to execute the program and without it, we cannot execute a program.

But it is not mandatory or necessary in the python language, we can execute a python program with or without the use of the main function.
1. Write the name of all the functions in this program

2. What are the arguments and return values of each function?

3. When do we need to use the return command?

Example

```python
import math

# howdy function
def howdy():
    print("Howdy!")

# welcome function
def welcome(name):
    print(f"Welcome {name}")
    print("Let's calculate the volume of a cylinder!")

# cylinder_volume function
def cylinder_volume(radius, height):
    area = math.pow(radius, 2) * math.pi
    volume = area * height
    return (volume)

# main function
def main():
    name = input("What is your name? ")
    howdy()
    welcome(name)
    radius = float(input("What is the radius of the cylinder? "))
    height = float(input("What is the height of the cylinder? "))
    volume = round(cylinder_volume(radius, height), 2)
    print(f"The volume of the cylinder is {volume}")

main()
```

The return command causes execution to leave the current function. It also allows the function to specify a return value to pass back to the code that called the function.
Functions: __name__ special variable

The interpreter sets a special variable __name__ to the value "__main__" when the program is executed by itself in a standalone way.

This helps us know whether the program is ran by itself or whether it is imported by another program.

Using this information, we may or may not execute some of the body of main().

Here is a scenario:
1. If we write multiple functions and a main function inside a program $P_1$
2. Then the functions in $P_1$ are imported by another program $P_2$
3. We would not want to execute the main function of $P_1$ in program $P_2$

Example

Write two Python programs.

Each program includes a main function.

Import the functions from the first program into the second program and use them.
Functions: `__name__` special variable

```python
if __name__ == "__main__":
    main()
```

1. The if statement checks the value of the `__name__` variable.
2. If the value is `"__main__"`, then the main function is called.
3. Otherwise, it is assumed that the program is imported into another program and we do not want to call main.

Note: conditional execution of the main function is useful when writing code that will be reused by other programs!

---

## Functions

Classify the functions based on input parameters and return values.

```python
def howdy():
    print("Howdy!")

def welcome(name):
    print(f"Welcome {name}"")
    print("Let's calculate the volume of a cylinder!")

def cylinder_volume(radius, height):
    area = math.pow(radius, 2) * math.pi
    volume = area * height
    return (volume)
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Input</th>
<th>Return Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>howdy()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>welcome(name)</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>cylinder_volume(radius, height)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Functions

Classify the functions based on input parameters and return values.

```python
def howdy():
    print("Howdy!")

def welcome(name):
    print(f"Welcome {name}")
    print("Let's calculate the volume of a cylinder!")

def cylinder_volume(radius, height):
    area = math.pow(radius, 2) * math.pi
    volume = area * height
    return (volume)
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Input</th>
<th>Return values</th>
</tr>
</thead>
<tbody>
<tr>
<td>howdy()</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>welcome()</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>cylinder_volume()</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Guessing game

Modify the The Guessing Game, by changing the part of the code to test if the number is too low or too high into a function.
import random

print('=================================================')
print('Welcome to The Guessing Game')
print('=================================================')

secret_num = random.randint(1, 100)
guess = int(input('Enter a number between 1 and 100: '))

attempts = 1

while guess != secret_num:
    if guess < secret_num:
        print('Too low.')
    else:
        print('Too high.')
    guess = int(input('Enter a number between 1 and 100: '))
    attempts += 1

print('Congratulations! You got it in %d guesses.' % (attempts))

import random

def test_number(guess, secret_num):
    if guess < secret_num:
        return 'Too low.'
    else:
        return 'Too high.'

print('=================================================')
print('Welcome to The Guessing Game')
print('=================================================')

secret_num = random.randint(1, 100)
guess = int(input('Enter a number between 1 and 100: '))

attempts = 1

while guess != secret_num:
    print(test_number(guess, secret_num))
    guess = int(input('Enter a number between 1 and 100: '))
    attempts += 1

print('Congratulations! You got it in', attempts, 'guesses.')
Variable scope

The **scope** of a variable is the **visibility** of the variable. It represents the regions of the program where the variable can be accessed and used.

Functions: local variables vs global variables

```python
v = 15  # global

def f1 ():
    v = 17  # local
    print ("v(f1): ", v)  # local
    v += 1  # local
    print ("v(f1): ", v)  # local

def f2 ():
    print ("v(f2): ", v)  # global

f1 ()
f2 ()
print ("v: ", v)  # global

>>> [evaluate variablesGL.py]
v(f1):  17
v(f1):  18
v(f2):  15
v:  15
```
Variables scope: local vs global

Global variables
Declared outside any function, and they can be accessed (used) on any function in the program.

Local variables
Declared inside a function, and can be used only inside that function.

Functions: local variables vs global variables

```python
v = 15  # global

def f1 ():
    v = 17  # local
    print ('v(f1):', v)  # local
    v += 1  # local
    print ('v(f1):', v)  # local

def f2 ():
    print ('v(f2):', v)  # global

f1 ()
f2 ()
print ('v:', v)  # global
```

- When we define a variable inside a function, it is local to the function.
- Any changes to this variable has no effect on similar variables outside of the functions.
Functions: local variables vs global variables

- A global variable is declared outside of the functions, and can be accessed by any functions in the program.

- A variable that is changed or created inside a function is local if it is not declared as a global variable.

- A variable cannot be local and global inside a function.

- To tell the interpreter that we want to use a global variable, we explicitly state it in the function using the global keyword.

Example

```python
v = 15

def f1 ():
    v = 17
    print ('v(f1): ', v)

def f2 ():
    v += 10
    print ('v(f2): ', v)

f1 ()
f2 ()
print ('v: ', v)
```

This program contains an error.
Find it and explain why it is an error.
Example

```python
v = 15

def f1 ():
v = 17
    print ('v(f1): ', v)

def f2 ()�:
global v
    v += 10
    print ('v(f2): ', v)

f1 ()
f2 ()
print ('v:', v)
```

To use the variable v in f2, declare v as global

```
-> [evaluate variablesGL3.py]
v(f1):  17
v(f2):  25
v:  25
```

Functions: local variables vs global variables

```python
def foo (x, y):
global a
    a = 42
    x, y = y, x
    b = 33
    b = 17
    c = 100
    print (a, b, x, y)
a, b, x, y = 1, 15, 3, 4
foo (17, 4)
print (a, b, x, y)
```

```
-> [evaluate variablesGL5.py]
42  17  4  17
42  15  3  4
```
A function can modify a global Variable.

```python
# I added a comment.
enemy_threat = False; # global variable

def do_something():
    # variables in functions are local by default. Set threat to be a global variable.
    global enemy_threat
    enemy_threat = True

def main():
    do_something()

    # Programmer still expects threat to be False.
    # But do_something() changed it to True!
    if enemy_threat:
        print('Launching nuclear missiles...')
    else:
        print('No threat detected.')
main()
```

- Use global variables with caution.
- It is tempting to use lots of global variables because they are easy to work with - especially when many functions are involved.
- Because every function has access to global variables, it can be hard to find which functions read and write these variables.
- To understand how the program works, we have to check every function that modifies the global state.
- Use local variables because other functions cannot affect them directly.
When we call a function with values, the values are assigned to the arguments according to their position.

The arguments can also be assigned via keywords.

```python
# Function Remainder
def remainder(number, divisor):
    return number % divisor

print('Remainder 20 / 7 : %d' % (remainder(20, 7)))

a = 22
b = 5
print('Remainder %d / %d : %d ' % (a, b, remainder(a, b)))

number = 36
divisor = 12
print('Remainder %d / %d : %d ' % (divisor, number, remainder(divisor, number)))
```

-> [evaluate teste.py]
Remainder 20 / 7 : 6
Remainder 22 / 5 : 2
Remainder 12 / 36 : 12
**Functions: arguments keyword**

```python
# Function Remainder
def remainder(number, divisor):
    return number % divisor

print('Remainder 20 / 7: %d' % (remainder(20, 7)))
print('Remainder 7 / 20: %d' % (remainder(7, 20)))
print('Remainder 20 / 7: %d' % (remainder(20, divisor = 7)))
print('Remainder 20 / 7: %d' % (remainder(number = 20, divisor = 7)))
print('Remainder 20 / 7: %d' % (remainder(divisor = 7, number = 20)))
```

```bash
[evaluate 'teste.py]
Remainder 20 / 7: 6
Remainder 7 / 20: 7
Remainder 20 / 7: 6
Remainder 20 / 7: 6
Remainder 20 / 7: 6
```

---

**68 Built-in Functions**

<table>
<thead>
<tr>
<th>abs()</th>
<th>divmod()</th>
<th>input()</th>
<th>ord()</th>
<th>type()</th>
</tr>
</thead>
<tbody>
<tr>
<td>all()</td>
<td>enumerate()</td>
<td>int()</td>
<td>pow()</td>
<td>vars()</td>
</tr>
<tr>
<td>any()</td>
<td>eval()</td>
<td>isinstance()</td>
<td>print()</td>
<td>zip()</td>
</tr>
<tr>
<td>ascii()</td>
<td>exec()</td>
<td>issubclass()</td>
<td>property()</td>
<td><strong>import</strong></td>
</tr>
<tr>
<td>bin()</td>
<td>filter()</td>
<td>iter()</td>
<td>range()</td>
<td></td>
</tr>
<tr>
<td>bool()</td>
<td>float()</td>
<td>len()</td>
<td>repr()</td>
<td></td>
</tr>
<tr>
<td>bytearray()</td>
<td>format()</td>
<td>list()</td>
<td>reversed()</td>
<td></td>
</tr>
<tr>
<td>bytes()</td>
<td>frozenset()</td>
<td>locals()</td>
<td>round()</td>
<td></td>
</tr>
<tr>
<td>callable()</td>
<td>getattr()</td>
<td>map()</td>
<td>set()</td>
<td></td>
</tr>
<tr>
<td>chr()</td>
<td>globals()</td>
<td>max()</td>
<td>setattr()</td>
<td></td>
</tr>
<tr>
<td>classmethod()</td>
<td>hasattr()</td>
<td>memoryview()</td>
<td>slice()</td>
<td></td>
</tr>
<tr>
<td>compile()</td>
<td>hash()</td>
<td>min()</td>
<td>staticmethod()</td>
<td></td>
</tr>
<tr>
<td>complex()</td>
<td>help()</td>
<td>next()</td>
<td>str()</td>
<td></td>
</tr>
<tr>
<td>delattr()</td>
<td>hex()</td>
<td>object()</td>
<td>sum()</td>
<td></td>
</tr>
<tr>
<td>dict()</td>
<td>id()</td>
<td>oct()</td>
<td>super()</td>
<td></td>
</tr>
<tr>
<td>dir()</td>
<td>input()</td>
<td>open()</td>
<td>tuple()</td>
<td></td>
</tr>
</tbody>
</table>