What is a Function?

- A **function** is a **named sequence of statements** that work together.
- Purpose: to organize programs into chunks that match how we think about solving a problem.
- Functions promote **abstraction**, **reuse**, and **readability**.

- `def` → keyword to define a function.
- `name` → function name (must follow identifier rules, not be a keyword).
- `parameters` → input values required by the function (may be empty).

Parameters & Arguments

- **Formal parameters** → variables in the function definition.
- **Arguments (actual parameters)** → values provided by the user when calling the function.
- Example:

Docstrings

- A string placed immediately after the function header.
- Used for **documentation** and retrievable with `function_name.__doc__`.

- Example:

```
def drawSquare(t, sz):
    """Make turtle t draw a square with side sz."""
```

Function Invocation (Calling)

- Defining a function does **not** run it.
- To execute: use a **function call** with parentheses.

Key Points

- **Some functions take multiple arguments**:
  - `math.pow(2, 3)` → base and exponent.
  - `max(7, 11, 4, 1, 17)` → returns the largest.
- **Return values**:
  - Functions like `abs`, `pow`, `max`, `range`, `int` return values that can be reused.
  - These are called **fruitful functions**.
- **Non-fruitful functions**:
  - Example: `drawSquare()` just draws a shape, does not return a value.

Writing your own fruitful functions:

```
def square(x):
    y = x * x
    return y
```

-

    - Uses `return` to give back a result.
    - Temporary variables like `y` help debugging (they are local variables).
- **Return vs Print**:
    - `return` gives back a value to the caller.
    - `print` only displays it, but does not return it.

  Example mistake:

```
def square(x):
    print(x * x)   # Wrong if you need the value
```

    - → This will make the function return `None`.
- **Execution flow**:
    - Defining a function (`def`) only saves its body; code runs only when the function is called.
    - After `return`, the function ends immediately—no later lines run.
- **Common errors**:
    - Having code after `return` (it never executes).
    - Printing instead of returning when a value is expected.

Unit Testing Overview:

- **Test Case:** Expresses a requirement of a program and can be checked automatically.
- **Unit Test:** An automated check to ensure a small unit of code (like a function) works correctly.
- **Test Suite:** A collection of multiple unit tests.

- Forces concrete thinking about program behavior.
- Provides automated feedback during development.
- Helps catch errors early, especially in large projects.

Python `assert` for Unit Testing:

- Syntax: `assert <expression>`
    - If the expression is **True**, nothing happens.
    - If **False**, a runtime error is raised.
- Useful for checking assumptions about data types, return values, or program state.

  Example:

```
assert type(9//5) == int
assert type(9.0//5) == int  # This fails because result is float
```

-

Using `assert` in Loops:

- Ensures all elements meet a condition.

```
lst = ['a','b','c']
first_type = type(lst[0])
for item in lst:
    assert type(item) == first_type
```

Return Value Tests:

- Simple way to test functions: compare function output to expected value.

Example:

```
def square(x):
    return x*x
assert square(3) == 9
```

- Important to choose representative inputs; testing all possible inputs is rarely feasible.

Key Points:

- `assert` is mainly for detecting failed assumptions early.
- Passing tests produce no output—only failures are flagged.
- More advanced testing can use frameworks like `unittest` for better reporting.
- Test cases are essential for debugging and maintaining code reliability.

Local Variables in Functions:

- Variables assigned inside a function are **local**; they exist only while the function runs.

Example:

```
def square(x):
    y = x * x
    return y
z = square(10)
```

- The **lifetime** of a local variable is limited to the function execution. When the function returns, the local variables are destroyed.

Formal Parameters Are Local:

- Function parameters act like local variables. Each call creates new local copies; previous values are not retained.

Global Variables:

- Functions can access global variables, but it's discouraged.

  Example of bad practice:

```
power = 2

def badsquare(x):

    y = x ** power  # uses global variable

    return y
```

- Recommended approach: pass global values as parameters instead of relying on them inside the function.

Local vs. Global Variables (Shadowing):

- Assigning a local variable with the same name as a global variable **shadows** the global variable. The local version is used within the function.

  Example:

```
power = 3

def powerof(x, p):

    power = p  # local variable shadows global 'power'

    return x ** power
```

- This prevents unintended changes to global variables but can confuse beginners.

Assignment to Parameters:

- Assigning a new value to a parameter inside a function changes only the local copy, not the variable in the caller's scope.

Key Points:

- **Scope:** the part of code where a variable can be accessed.
- **Local variable:** exists only within the function; temporary.
- Using the same name for local and global variables is allowed but discouraged.
- Best practice: pass values explicitly as parameters and avoid modifying globals inside functions.