



CMSC 105 Elementary Programming

Acknowledgement: These slides are adapted from slides provided with "Introduction to Programming Using Python, Liang (Pearson 2013)" and slides shared by Dr. Jory Denny

Functions

Outline

Opening Problem

- Find the sum of integers from 1 to 10, from 20 to 30, and from 35 to 45, respectively.
- Compute the square root of a number over and over again
- Organize a large program into smaller components

Need of reusable and simplified code!

Problem

```
1. sum = 0
2. for i in range(1, 11):
3.     sum += i
4. print("Sum from 1 to 10 is", sum)
```

```
1. sum = 0
2. for i in range(20, 30):
3.     sum += i
4. print("Sum from 20 to 30 is", sum)
```

```
1. sum = 0
2. for i in range(35, 46):
3.     sum += i
4. print("Sum from 35 to 45 is", sum)
```

Solution

```
1. def sum(i1, i2):  
2.     res = 0  
3.     for i in range(i1, i2+1):  
4.         res += i  
5.     return res  
6.
```

Function
Definition

```
7. def main():  
8.     print("Sum from 1 to 10 is ", sum(1, 10))  
9.     print("Sum from 20 to 30 is ", sum(20, 30))  
10.    print("Sum from 35 to 45 is ", sum(35, 45));
```

Function
Invocation

```
11. # Invoke the main Function  
12. if __name__ == '__main__':  
13.     main()
```

Main is also a function. While we are by no means required to provide a function called main, it is convention. Other languages require such constructs.

Python built-in functions

For example, you've used a few of Python's built-in functions already:

print

input

len

int

random.randint

Python built-in functions

For example, you've used a few of Python's built-in functions already:

print

input

len

int

random.randint

You don't need to know how they work. You just use them and they do work.

User-defined functions

No language has all the built-in functions that you'd ever need, so every language gives you the ability to **create your own functions**.

It's necessary to make your own functions in order to write **well-designed programs**.

Function Definitions


Format

```
def name (parameters) :  
    body
```

- The **name** can be any valid identifier.
- The **parameters** are the unknowns that you use in the body, maybe none at all.
- The **body** can be any group of statements.

Example 1

```
>>> def print_hello():  
    print("Hello there!")
```



No formal parameters

```
>>> print_hello()  
Hello there!
```



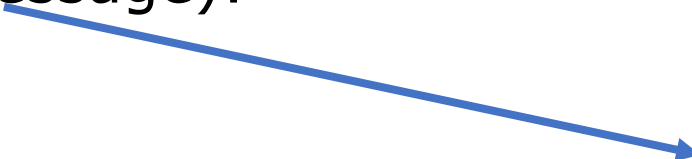
- Calling the function
- No parameter value passed
 - No return value

What will be the output?

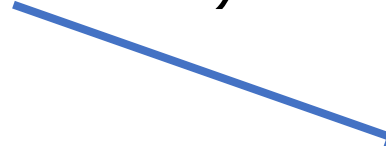
Example 2

```
>>> def print_hello(message):  
    print(message)
```

```
>>> print_hello("Hi there!")  
Hi there!
```



Formal parameter stored in variable message.

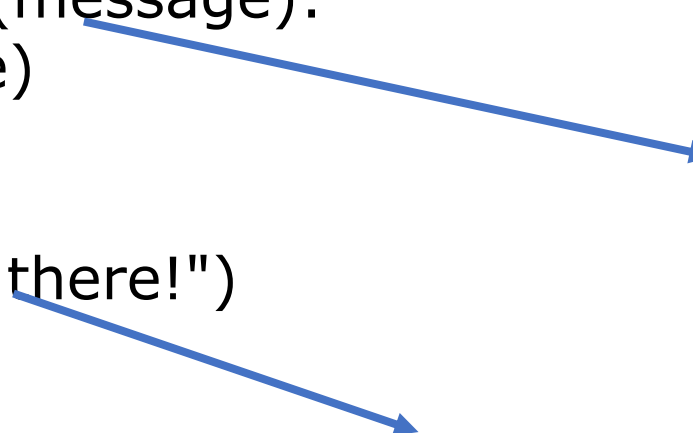


Actual value of the variable while calling function.

Example 2

```
>>> def print_hello(message):  
    print(message)
```

```
>>> print_hello("Hi there!")  
Hi there!
```



The name variable is undefined right now, but when the function is called, message will receive the value of the argument.

An argument is required here when you call the function...

Example 2

```
>>> def print_hello(message):  
    print(message)
```

```
>>> print_hello(message)
```



Is it correct?

Traceback (most recent call last):

File "<pyshell#8>", line 1, in <module>
 print_hello(message)

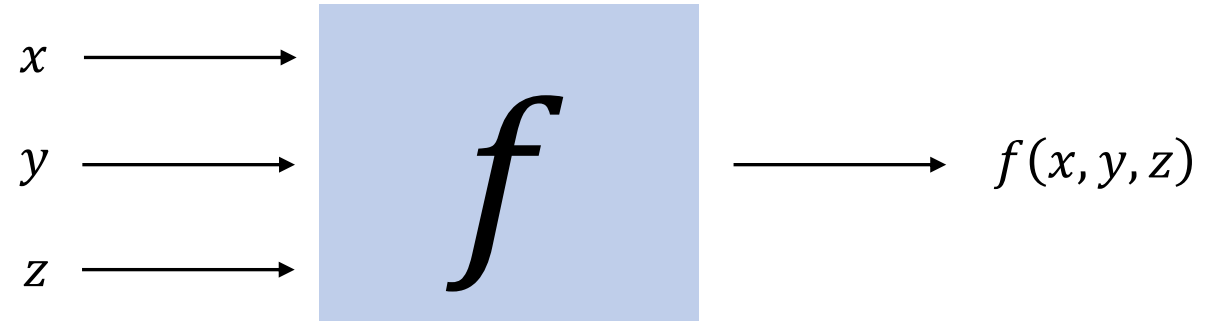
NameError: name 'message' is not defined

Why do you see this error?

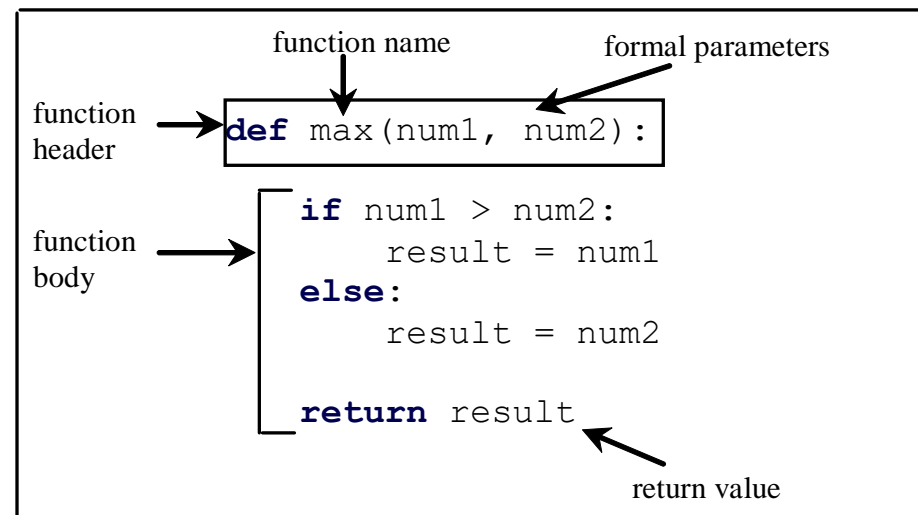
Defining Functions

- A **function** is a collection of statements that are grouped together to perform an operation.

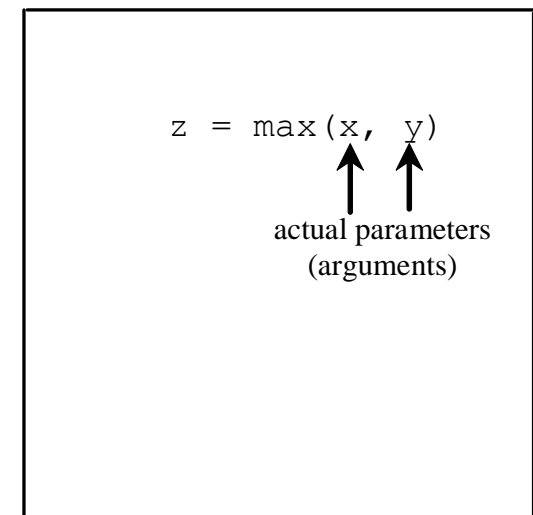
- "function"
- "subroutine"
- "algorithm"



Define a function

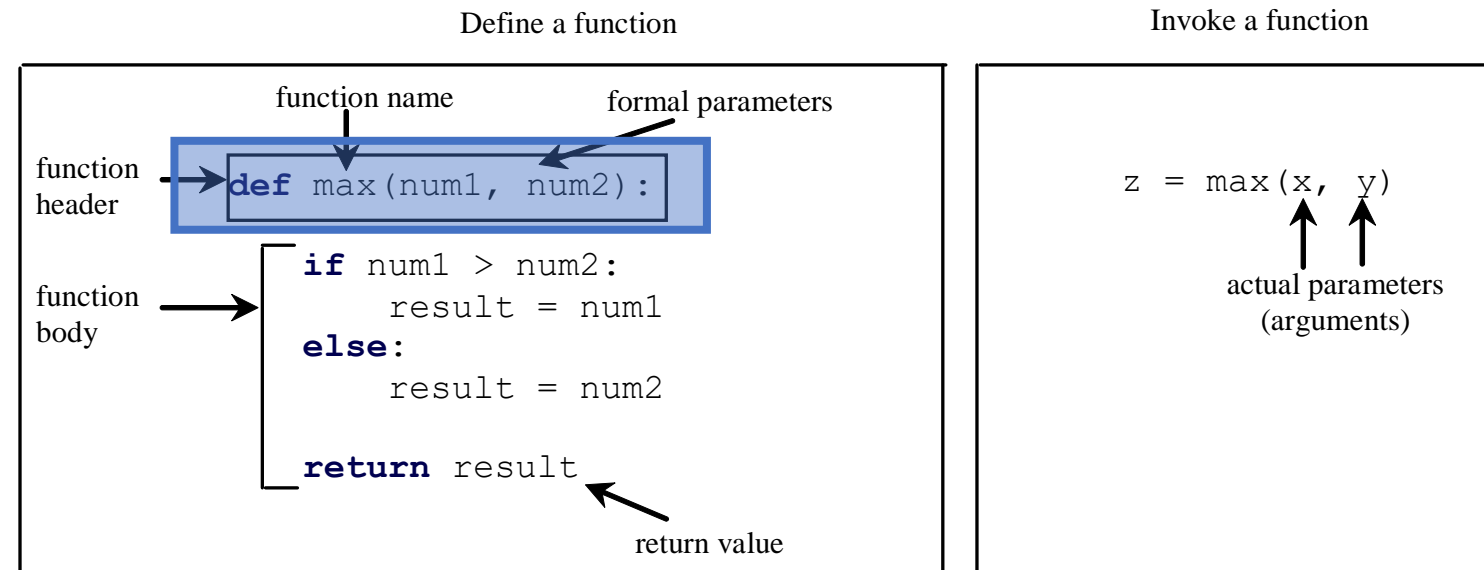


Invoke a function



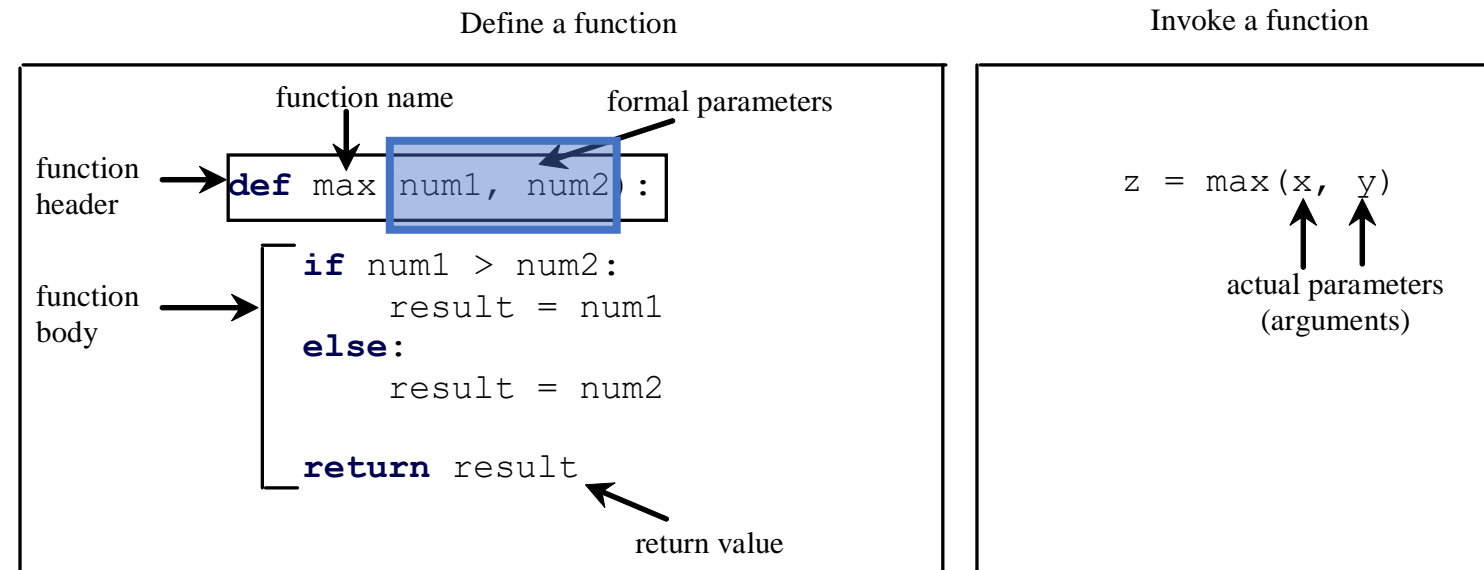
Function Header

- A function contains a **header** and **body**. The header begins with the **def** keyword, followed by function's name and parameters, followed by a colon.



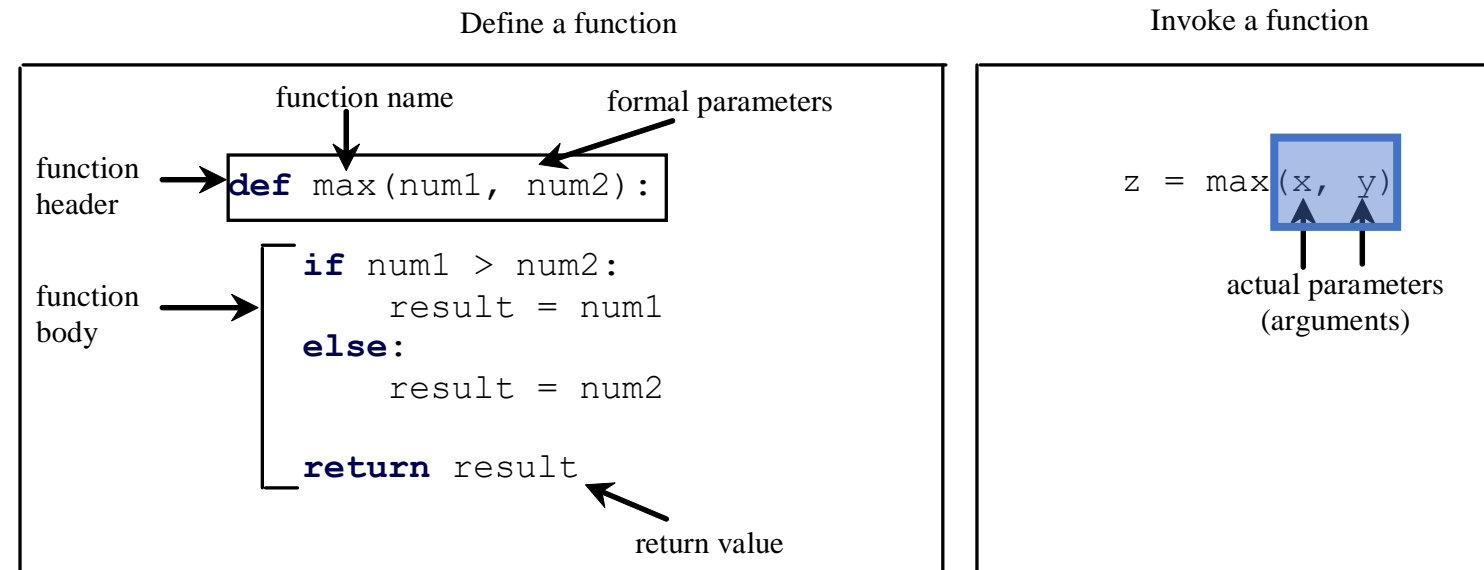
Formal Parameters

- The variables defined in the function header are known as **formal parameters**.



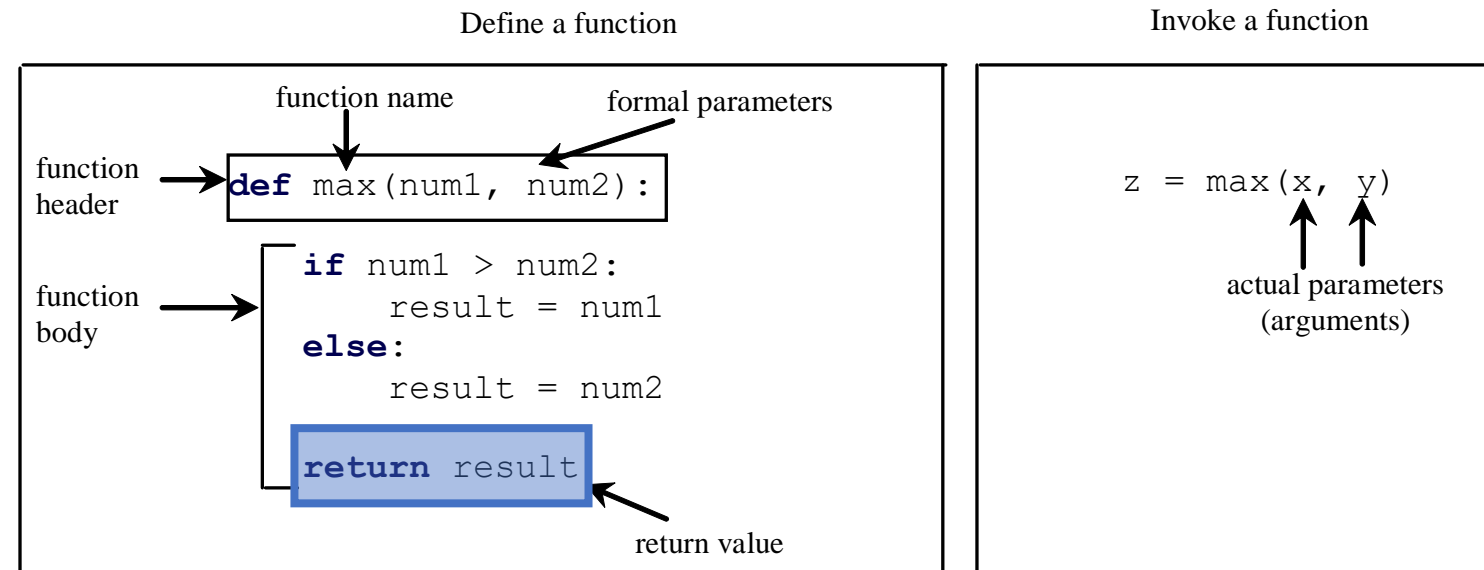
Actual Parameters

- When a function is **invoked**, you pass a value to the parameter. This value is referred to as **actual parameter** or **argument**.



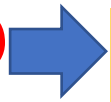
Return Value

- A function optionally may **return** a value using the **return** keyword.



Example

```
>>> def find_sum(num1,num2):  
    sum_val=num1+num2  
    print(sum_val)
```



Prints the
sum

```
>>> find_sum(2,3)
```

5

```
>>> def find_sum(num1,num2):  
    sum_val=num1+num2  
    return sum_val
```



```
>>> value=find_sum(2,3)
```

```
>>> value
```

5

Value stored in
variable 'value'
can be used
later in the
program

find_sum
returns the
result and
stores it in
variable 'value'

Exercise

- Write a function that takes as **input parameter** a number and **returns** boolean value True/False if the number is/is not an even number.
- Write a function that takes as input a number, let's say **num** and returns the incremented value **(num+1)**

Function Details

Return Values

- Functions do not need return values/statements

- Example:

```
def doSomething() :  
    print("Hi there")
```

Try it with this program.
Invoke
`print(doSomething())`.

- In this case, functions automatically return a special value in Python – `None`.
 - It is a keyword, similar to `True` and `False`.
 - Other languages refer to this as "void" and it is not actually a value

Passing Arguments by Positions

- Consider:

```
def nPrintln(message, n):  
    for i in range(n):  
        print(message)
```

- What happens with the following invocations?

- `nPrintln("Hi", 5)`
- `nPrintln("Class", 2)`
- `nPrintln(4, "What now?")`



Type error!

Passing Arguments by Keywords

- Consider:

```
def nPrintln(message, n):  
    for i in range(n):  
        print(message)
```

- What about the following?
 - `nPrintln(n=4, message="What now?")`
- This is completely ok and normal in Python

Passing Variables

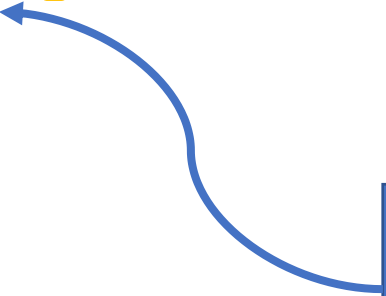
- In python, all data are **objects** and variables are actually a **reference** to an object
- When you invoke a function with arguments, the reference value of each argument is passed to the parameter. This is referred to as **pass-by-value**.
- The value is actually a reference value to the object



Reuse Functions from Other Files

- One of the benefits of functions is for reuse.
- Simply import, and use the Function
- Example

```
from math import sqrt  
sqrt()
```



From states the file from which a function or class is taken. Import brings the name into the program

Scope

- **Scope** is the part of the program where a variable can be referenced
- A variable created inside a function is referred to as a **local variable**.
 - Local variables can only be accessed inside a function.
 - The scope of a local variable starts from its creation and continues to the end of the function that contains the variable.
- In Python, you can also use **global variables**.
 - They are created outside all functions and are accessible to all functions in their scope.
 - You have been using these exclusively until now

Scope Example 1

```
globalVar = 1
def f1():
    localVar = 2
    print(globalVar)
    print(localVar)
f1()
print(globalVar)
# Out of scope. This gives an error
print(localVar)
```

What is output?

Scope Example 2

```
x = 1
def f1():
    x = 2
    # Displays 2
    print(x)
f1()
# Displays 1
print(x)
```

What is output?

Scope Example 3

```
x = eval(input("Enter a number: "))  
if x > 0:  
    y = 4  
# Gives an error only if y is not created  
print(y)
```

What is output?

Scope Example 4

```
sum = 0
for i in range(5):
    sum += i
# Displays 4
print(i)
```

What is output?

Scope Example 5

```
x = 1
def increase():
    global x
    x += 1
    # Display 2
    print(x)
increase()
# Display 2
print(x)
```

What is output?

Default arguments

- You are allowed to define default arguments for parameters
- When the function is invoked without the parameter, the default value is used
- Example

```
def incr(n, i=1):  
    return n + i
```

```
x = 1
```

```
x = incr(x, 4)
```

```
x = incr(x) # Invoked like incr(x, 1)
```

Multiple Return values

- Python also allows returning multiple values at a time. Example:

```
def sqrAndCube(x) :  
    sqr = x*x  
    cube = sqr*x  
    return sqr, cube  
sqr, cube = sqrAndCube(5)  
print(sqr, cube)
```

Exercise

- Write a function that takes as **input parameters** two numbers and **returns** them in ascending order.
- Write a function that calculates the area of a circle given its radius.

Use this expression: **`a = 3.14 * r ** 2`**

It should work like this:

```
>>> circle(2)  
12.56
```

Modularizing Code

- Functions can be used to reduce redundant coding and enable code reuse. Functions can also be used to modularize code and improve the quality of the program.
- Benefits of functions
 - Write a function once and reuse it anywhere.
 - Information hiding. Hide the implementation from the user.
 - Reduce complexity.

Software Development

- Things to remember
 - You rarely write code for yourself
 - Rather, you belong to a team working towards a common goal, where no one person can know everything of the code.
- How can we communicate intent of code and its design?
 - Documentation
- How do we develop large programs?
 - Stepwise refinement

Documentation

- We use comments to relay intent of control flow and difficult to understand statements
 - It is a fine balance between too much and too little commenting
- For larger control structures, i.e., functions, methods, and classes, we should provide official documentation to specify its use
 - We will use docstring format
 - Every class, method, and function will need a docstring describing its purpose, formal arguments, and return value.

Documentation

- Docstring example:

```
def square(n):
```

```
    """
```

```
    Square a number.
```

```
    Arguments:
```

```
        n: A number.
```

```
    Returns:
```

```
        The square of the input number.
```

```
    """
```

```
    return n*n
```

""" Denotes the start and end of a docstring. If you use the `help()` function in python, it prints the docstring. Try it with `help(square)`.

Always provide a brief explanatory statement.

If arguments are needed, document each one with a description. Otherwise do not have an "Arguments" section.

If the function returns a value, document its meaning in the "Returns" section.

Stepwise Refinement

- The concept of function abstraction can be applied to the process of developing programs.
- When writing a large program, you can use the "divide and conquer" strategy, also known as stepwise refinement, to decompose it into subproblems.
- The subproblems can be further decomposed into smaller, more manageable problems.

Benefits of Stepwise Refinement

- Simpler programs
- Reusing functions
- Easier developing, debugging, and testing
- Better for facilitating teamwork



Thank you!
Questions?