CHAPTER 6 FUNCTIONS

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH INTRODUCTION TO JAVA PROGRAMMING, LIANG (PEARSON 2014)

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

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

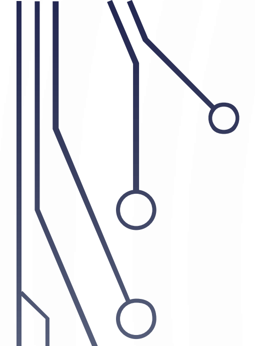
```
1.  def sum(i1, i2):
2.    res = 0
3.    for i in range(i1, i2+1):
4.      res += i
5.    return res
6.
7.  def main():
8.    print("Sum from 1 to 10 is ", sum( 1, 10));
9.    print("Sum from 20 to 30 is ", sum(20, 30));
10.   print("Sum from 35 to 45 is ", sum(35, 45));

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

Function Definition

Function Invocation

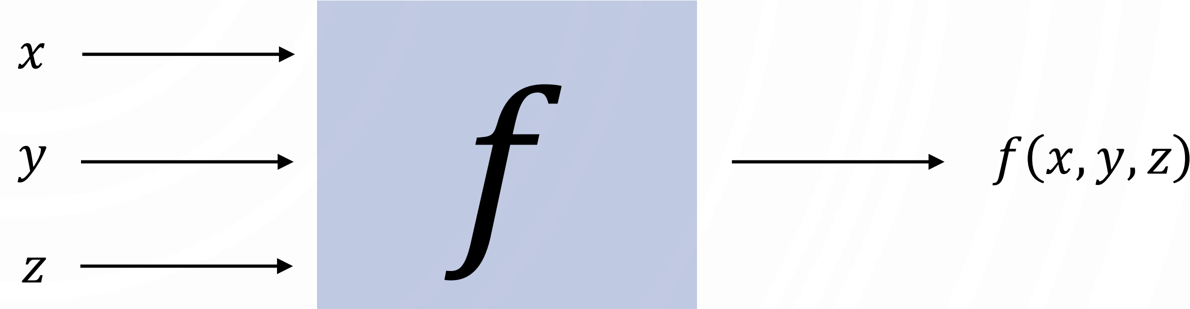
Main is also a function. While we are by no means required to provide a Function called main, it is convention. Other languages require such constructs.

FUNCTION DEFINITIONS

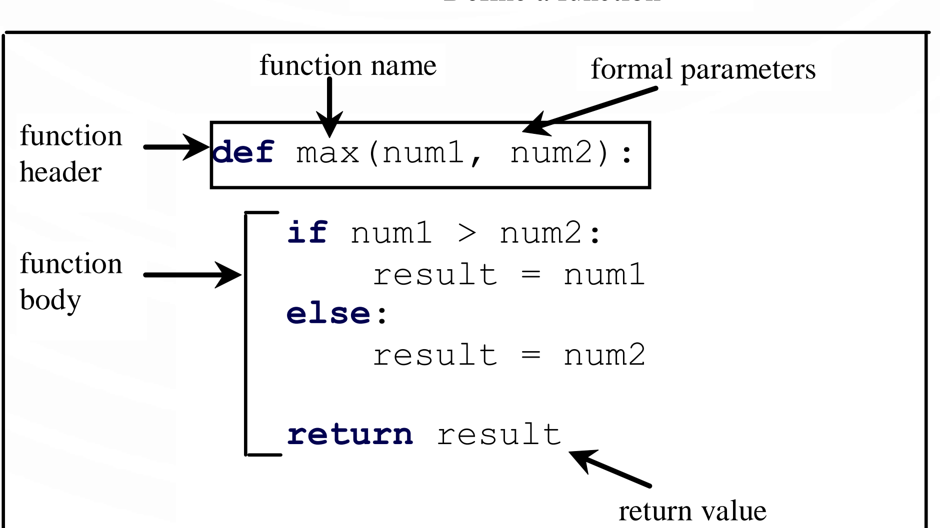


DEFINING FUNCTIONS

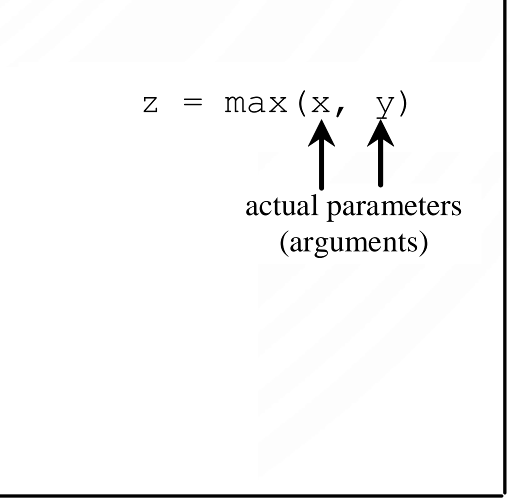
- A **function** is a collection of statements that are grouped together to perform an operation.
  - "function"
  - "subroutine"
  - "algorithm"

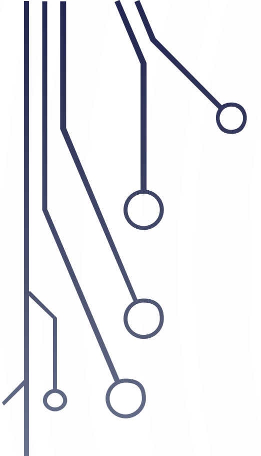


Define a function



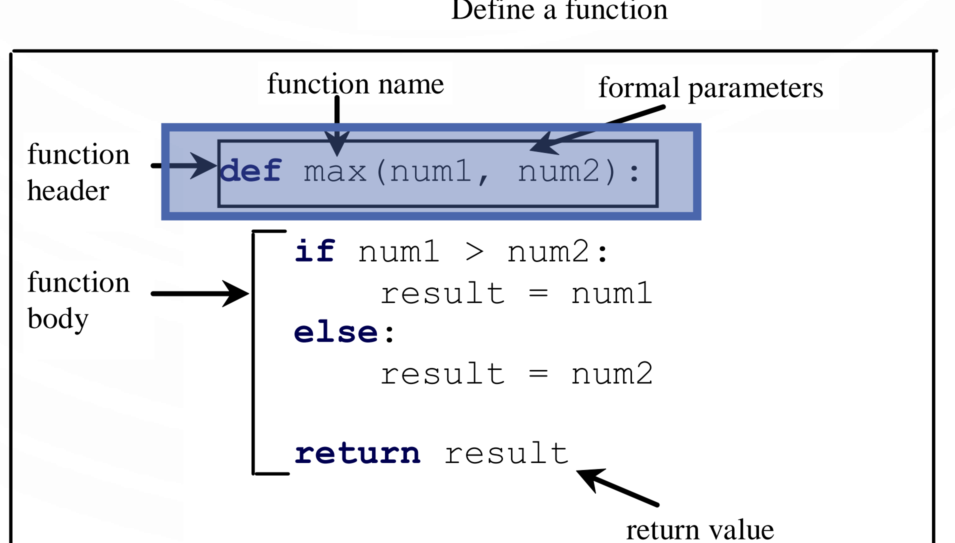
Invoke a function

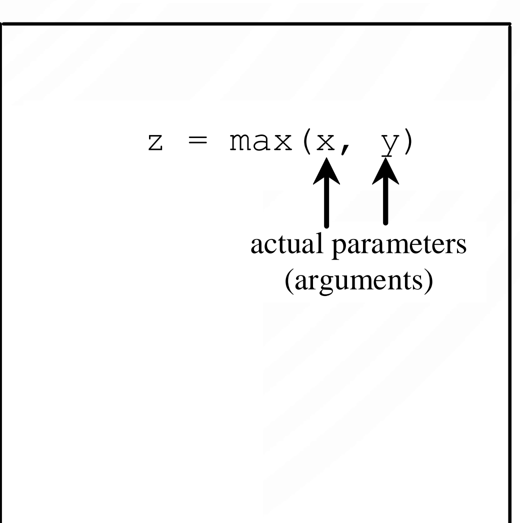


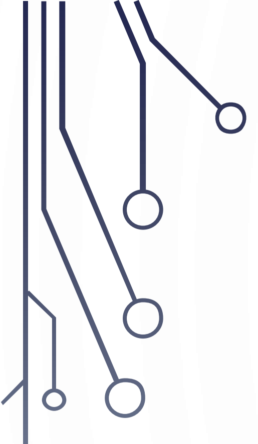


FUNCTION HEADER

- 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.

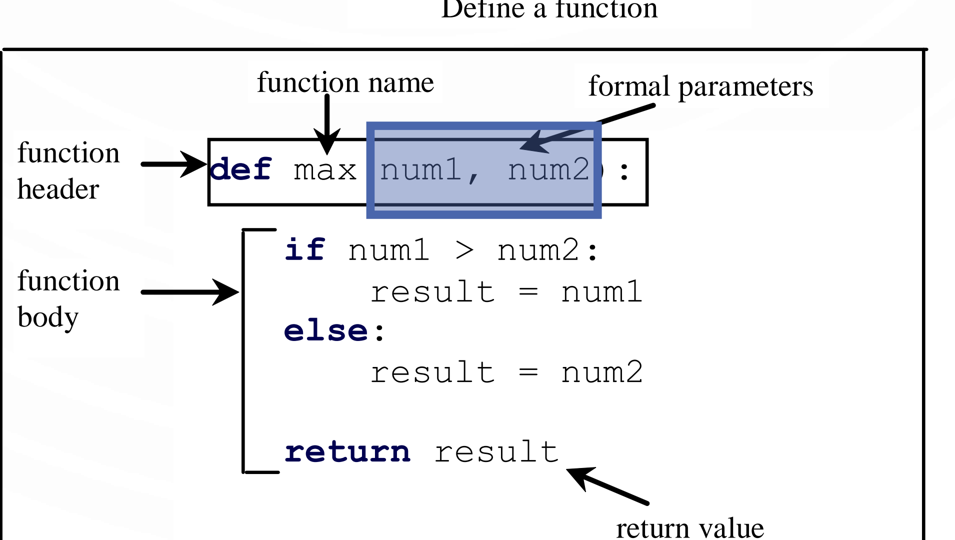


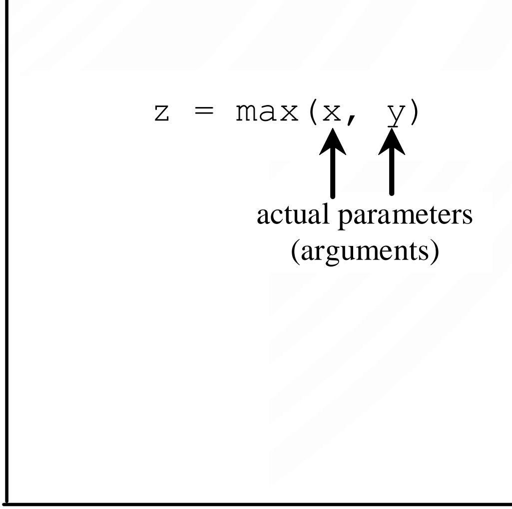




FORMAL PARAMETERS

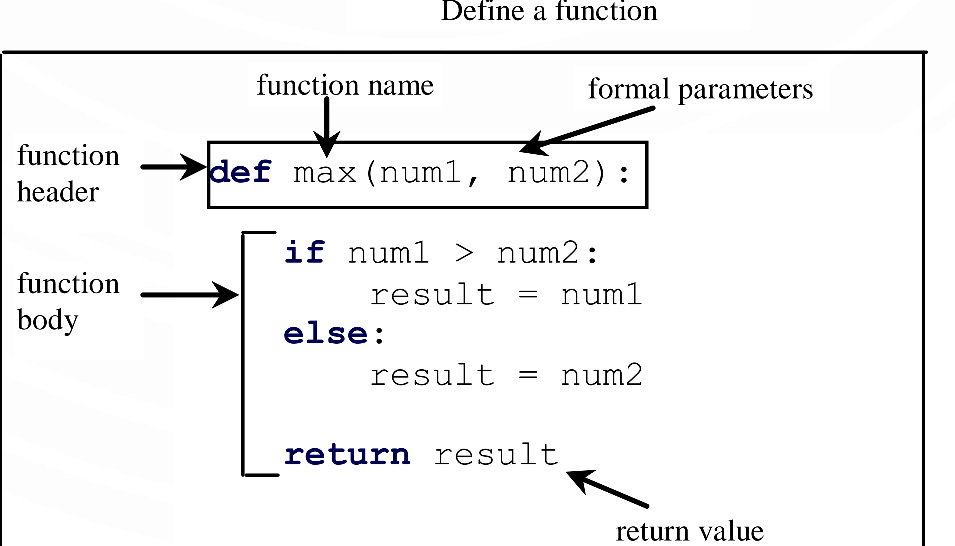
- The variables defined in the function header are known as **formal parameters**.

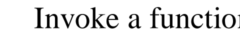


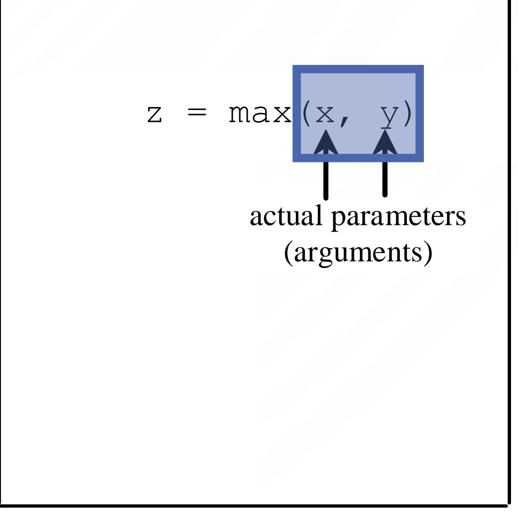


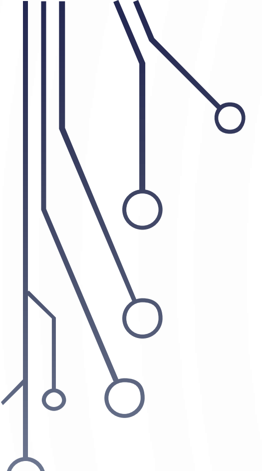
ACTUAL PARAMETERS

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



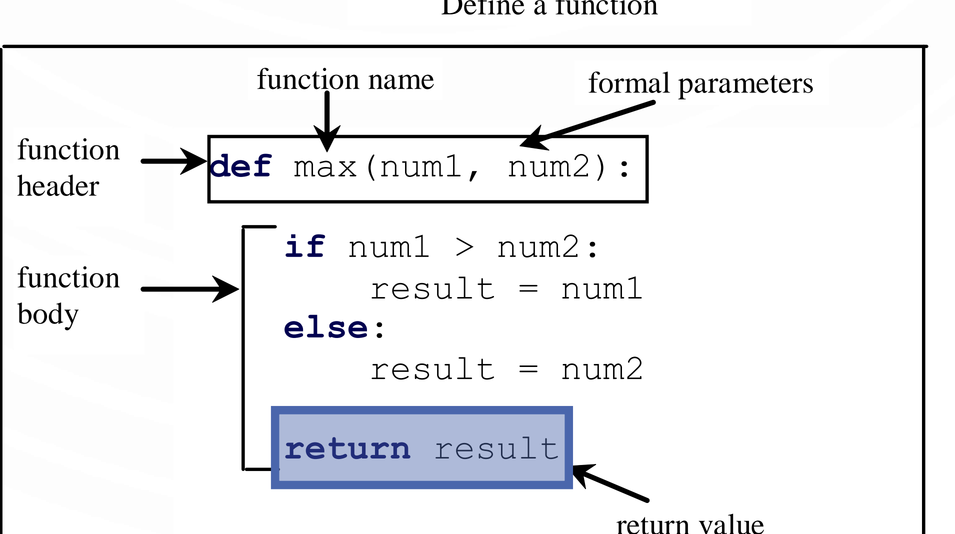


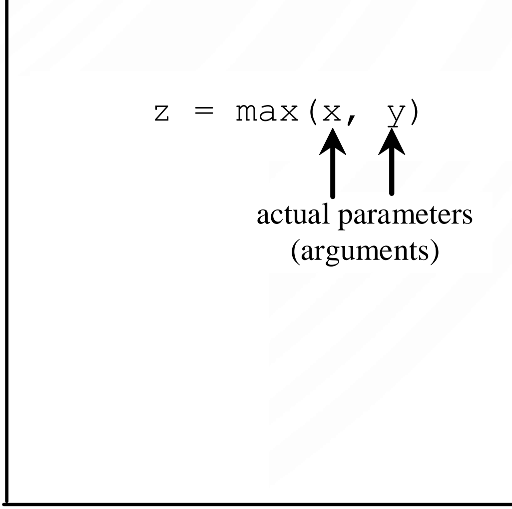




RETURN VALUE

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





EXAMPLE

- Lets write a function to compute a random integer between $[a, b]$
- Lets write a function to computes the square of a number
- Lets write a function to print a point in a special format
- What do you observe about Functions?

EXERCISE

- On the white boards
  - Write a function to determine if two circles overlap
  - Write a function that converts a number to binary in string form, e.g., if 13 was provided as input, then "1101" would be returned as output.

DEVELOP A ROBOT PROGRAM

- Spins in one complete circle at intervals of 10 degrees and scans for a bright light
    - Once a "bright" light is found, the robot moves forward 0.1 meters
- The robot should continue to operate until 5 successful scans are complete
- Turn the light on when the robot is moving forward (off otherwise)

- In this program write and use as many functions as possible to organize your solution. Starting with a main program. (*hint* -- my solution had about 4 functions)

FUNCTION TRACING

TRACING FUNCTION CALLS

```
if __name__ == '__main__':
    main()
```

Memory

```
def main():
    i = 5
    j = 2
    k = max(i, j)

    print("The maximum of", i, "and", j,
    "is", k)
```

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result
```

TRACING FUNCTION CALLS

```
if __name__ == '__main__':
    main()
```

Memory

Before main's memory space

```
def main():
    i = 5
    j = 2
    k = max(i, j)

    print("The maximum of", i, "and", j, "is", k)
```

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result
```

TRACING FUNCTION CALLS

```
if __name__ == '__main__':
    main()
```

```
def main():
    i = 5
    j = 2
    k = max(i, j)

    print("The maximum of", i, "and", j, "is", k)
```

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result
```

Memory

Main's memory space

Before main's memory space

TRACING FUNCTION CALLS

```
if __name__ == '__main__':
    main()
```

```
def main():
    i = 5
    j = 2
    k = max(i, j)

    print("The maximum of", i, "and", j, "is", k)
```

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result
```

Memory

Main's memory space
i: 5
j: 2

Before main's memory space

TRACING FUNCTION CALLS

```
if __name__ == '__main__':
    main()
```

```
def main():
    i = 5
    j = 2
    k = max(i, j)

    print("The maximum of", i, "and", j, "is", k)
```

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result
```

Memory

Max's memory space
num1: 5
num2: 2

Main's memory space
i: 5
j: 2

Before main's memory space

TRACING FUNCTION CALLS

```
if __name__ == '__main__':
    main()
```

```
def main():
    i = 5
    j = 2
    k = max(i, j)

    print("The maximum of", i, "and", j, "is", k)
```

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result
```

Memory

Max's memory space
num1: 5   result: 5
num2: 2

Main's memory space
i: 5
j: 2

Before main's memory space

TRACING FUNCTION CALLS

```
if __name__ == '__main__':
    main()
```

```
def main():
    i = 5
    j = 2
    k = max(i, j)

    print("The maximum
      of", i, "and", j,
      "is", k)
```

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result
```

Memory

Max's memory space
num1: 5 result: 5
num2: 2

Main's memory space
i: 5
j: 2
k: 5

Before main's memory space

TRACING FUNCTION CALLS

```
if __name__ == '__main__':
    main()
```

```
def main():
    i = 5
    j = 2
    k = max(i, j)

    print("The maximum of", i, "and", j,
          "is", k)
```

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result
```

Memory

Main's memory space
i: 5
j: 2
k: 5

Before main's memory space

Output

```
The maximum of 5 and
2 is 5
```

TRACING FUNCTION CALLS

```
if __name__ == '__main__':
    main()
```

```
def main():
    i = 5
    j = 2
    k = max(i, j)

    print("The maximum of", i, "and", j,
    "is", k)
```

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
    return result
```

Memory

Main's memory space
i: 5
j: 2
k: 5

Before main's memory space

Output
The maximum of 5 and 2 is 5

TRACING FUNCTION CALLS

```
if __name__ == '__main__':
   main()
```

Memory

Before main's memory space

```
def main():
   i = 5
   j = 2
   k = max(i, j)

   print("The maximum
      of", i, "and", j,
      "is", k)
```

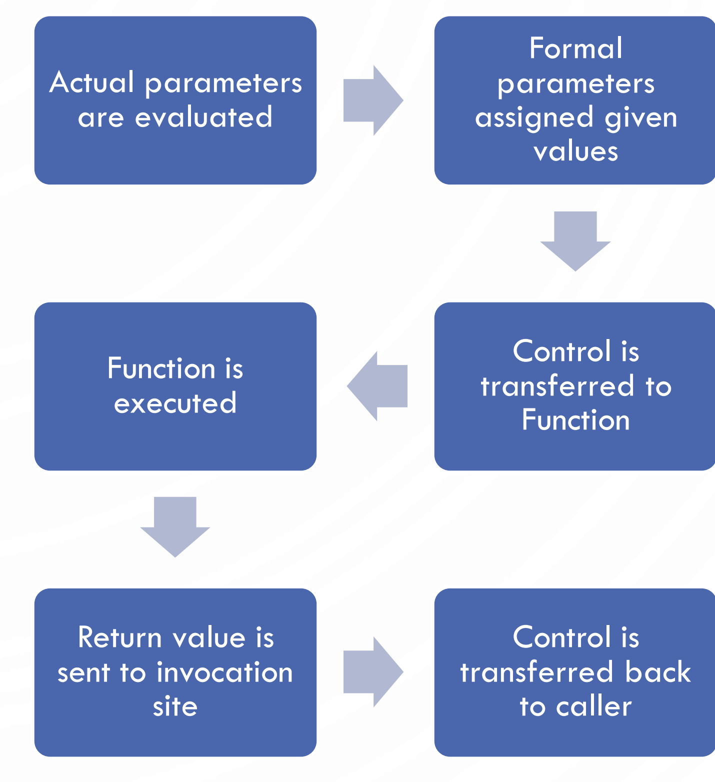
```
def max(num1, num2):
   if num1 > num2:
      result = num1
   else:
      result = num2
   return result
```

Output

The maximum of 5 and
2 is 5

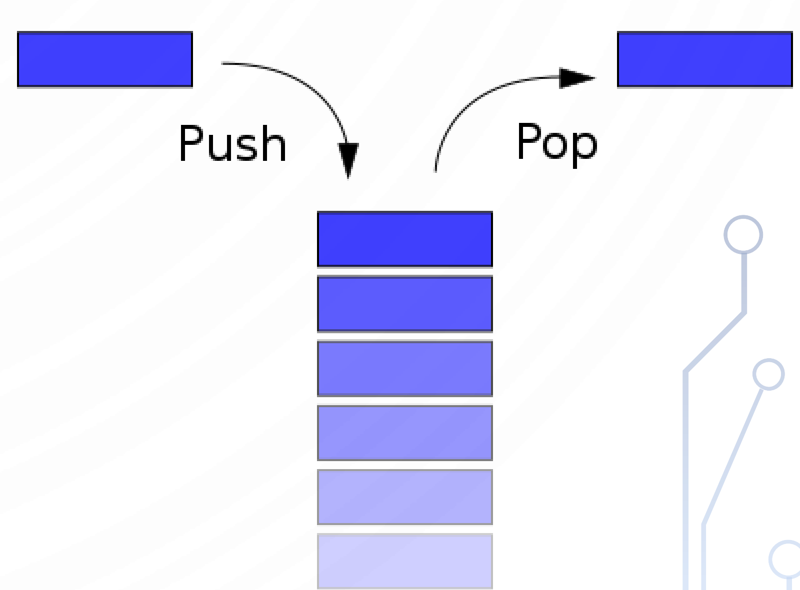
FUNCTION CONTROL FLOW

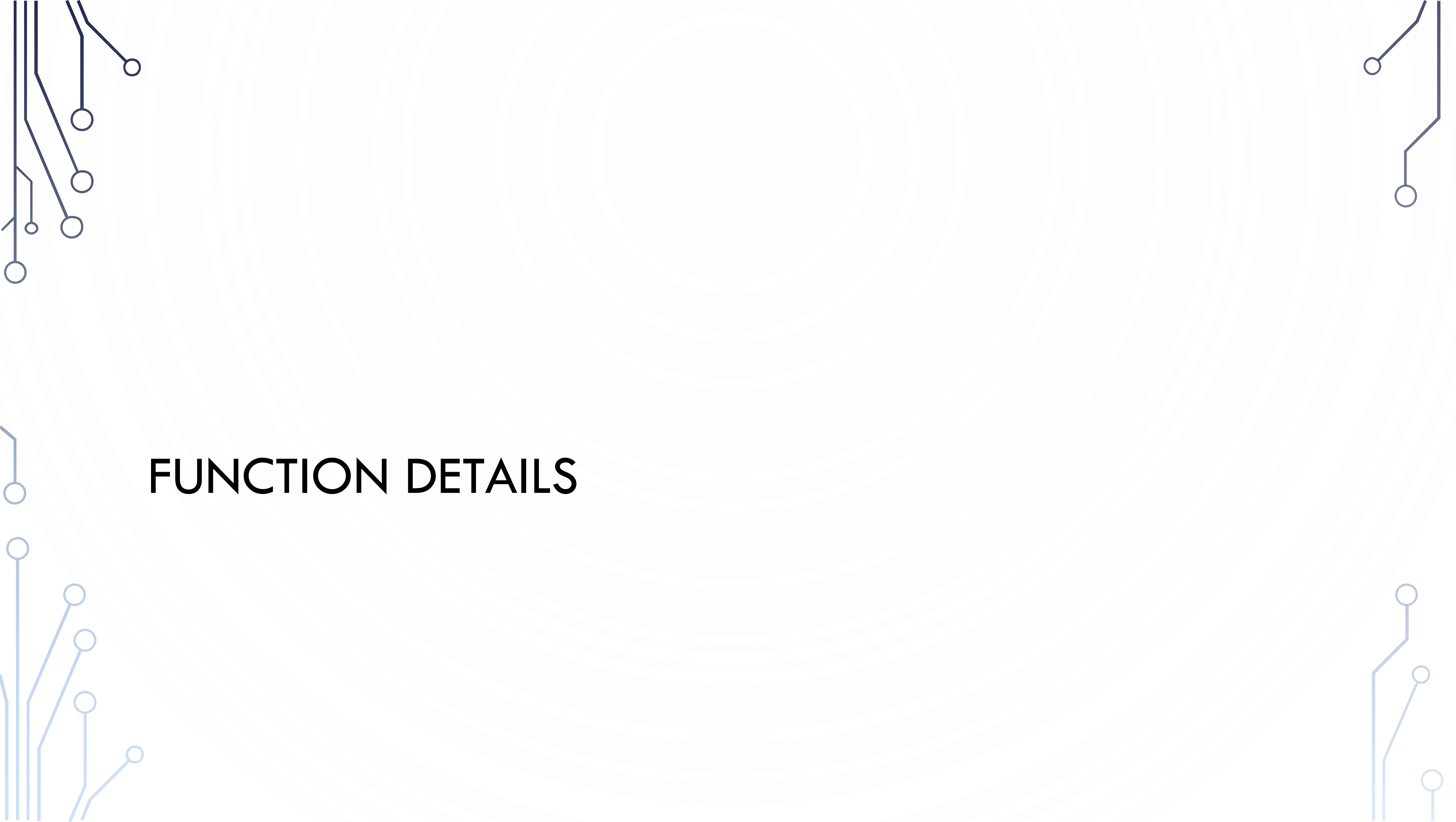
- Key point. Functions provide a new way to control the flow of execution.
- What happens when a function is called:



CALL STACKS

- The memory operates in the manner of a stack, called the **call stack**
  - Each function invocation has its own memory space on the top of the stack that is allocated when the function is called
  - After running the function and returning, this memory is released and variables of the function are no longer accessible
- Important – each time the function is invoked, a new space in memory is acquired and the variables are given new values




FUNCTION DETAILS

RETURN VALUES

- Functions do not need return values/statements
- Example:

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

- 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

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

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, a variables reference is passed into the function, i.e., Python is **pass-by-object-reference**.

Refers to

```
varName
otherVarName = varName
```

Object memory

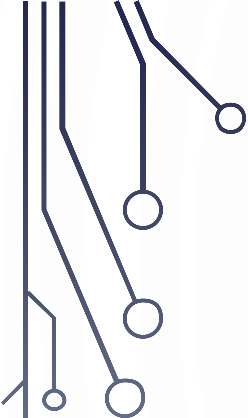
- For now and for simplicity, assume Python is ***pass-by-value***, essentially, meaning that changes to a variable inside the function do not affect the variable passed to it, i.e., it is copied
  - Numbers and strings in python work this way, but we will amend this rule when we learn more about objects

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 5
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

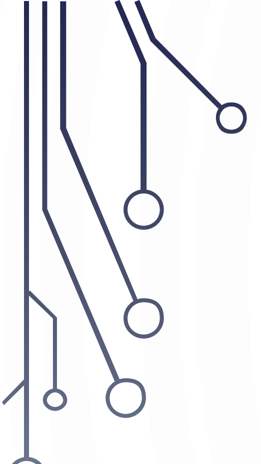
```
def incr(n, i=1):
  return n + i
x = 1
x = incr(x, 4)
x = incr(x) # Invoked like incr(x, 1)
```

MULTIPLE RETURN VALUES

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

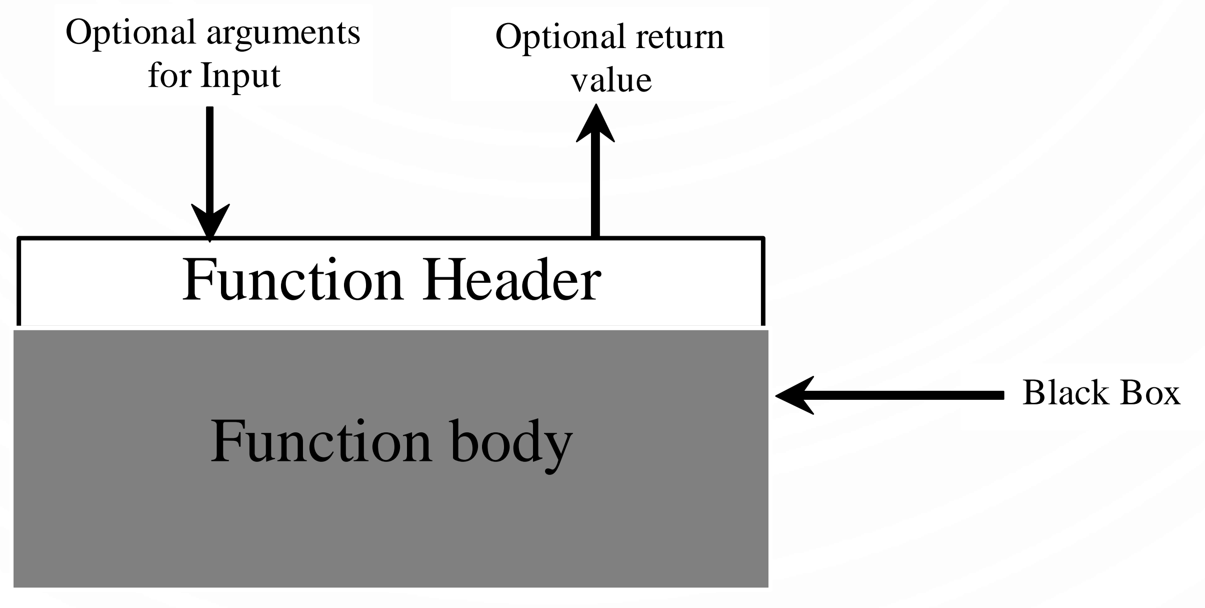
```
def sqrAndCube(x):
    sqr = x*x
    cube = sqr*x
    return sqr, cube
sqr, cube = sqrAndCube(5)
print(sqr, cube)
```

MODULARIZATION



FUNCTION ABSTRACTION

- You can think of the function body as a black box that contains the detailed implementation for the function.



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 rembember
  - 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.

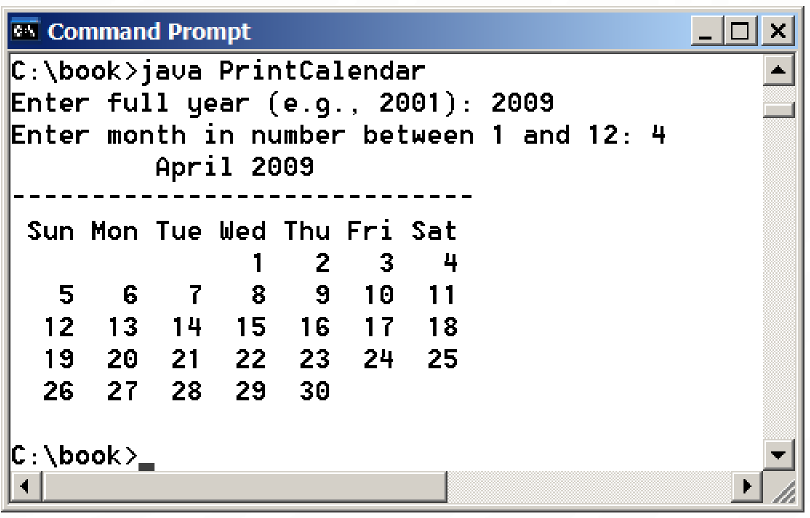
If the function returns a value, document its meaning in the "Returns" section.

STEPWISE REFINEMENT

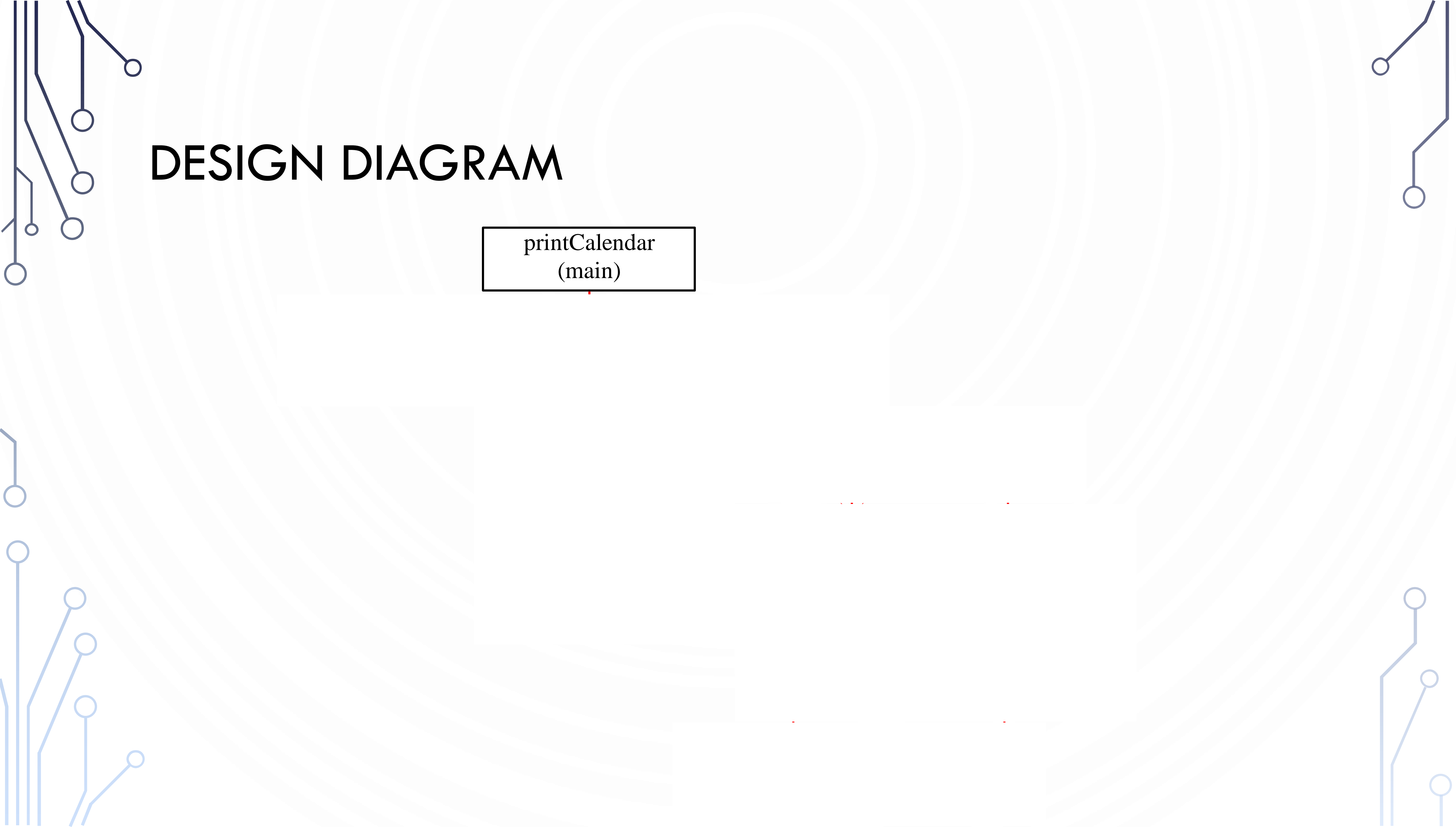
- The concept of function abstraction can be applied to the process of developing programs. When writing a large program, you can use the "divide and conquer" strategy, also known as stepwise refinement, to decompose it into subproblems. The subproblems can be further decomposed into smaller, more manageable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

PRINTCALENDER CASE STUDY

- Let us use the PrintCalendar example to demonstrate the stepwise refinement approach.



DESIGN DIAGRAM



IMPLEMENTATION: TOP-DOWN

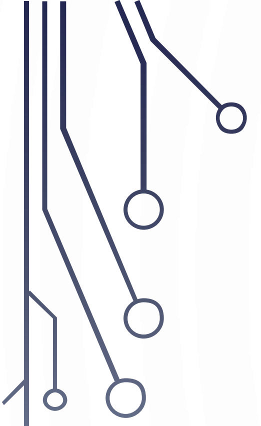
- Top-down approach is to implement one function in the structure chart at a time from the top to the bottom. Stubs can be used for the functions waiting to be implemented. A stub is a simple but incomplete version of a function. The use of stubs enables you to test invoking the function from a caller. Implement the main function first and then use a stub for the printMonth Function. For example, let printMonth display the year and the month in the stub. Thus, your program may begin like this:

IMPLEMENTATION: BOTTOM-UP

- Bottom-up approach is to implement one function in the structure chart at a time from the bottom to the top. For each function implemented, write a test program to test it. Both top-down and bottom-up functions are fine. Both approaches implement the functions incrementally and help to isolate programming errors and makes debugging easy. Sometimes, they can be used together.

BENEFITS OF STEPWISE REFINEMENT

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



UNIT TESTING

- **Unit test** – Automated piece of code that invokes a "unit" of work and then checks a single assumption of its behavior. Use main() to test each library.
- Follow SETT – unit testing paradigm
  - Setup – create data for input and predetermine the output
  - Execute – call the function in question
  - Test – analyze correctness and determine true/false for test
  - Teardown – cleanup any data, close buffers, etc

- Take a function to compute the square of a number. Here is a good unit test for that function:

```
def testSquare() {
  # Setup - predetermined values!
  x = 5
  ans = 25

  # Execute - call the function
  sqr = square(x)

  # Test
  return sqr == ans

  # Empty teardown
```

CLARIFICATION TIME!

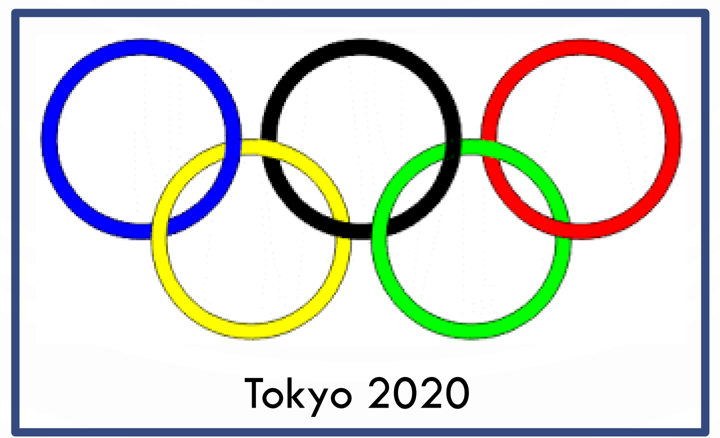
- Write any and all questions you have on an index card. It can range from asking about my experiences, specific things in Python, etc.

EXERCISE

- Financial Application: Loan Amortization Schedule.
  - The monthly payment for a given loan pays the principle and the interest.
    - The monthly interest is computed by multiplying the monthly interest rate and the balance (the remaining principle).
    - The principle paid for the month is therefore the monthly payment minus the monthly interest.
  - Write a program that lets the user enter the loan amount, number of years, and the annual interest rate, and then displays the amortization schedule for a loan in a formatted table.
    - Compute the monthly payment and total payment amounts
    - Monthly payment is computed by: $\frac{a}{((1+r)^n-1)/(r(1+r)\hat{}n)}$, where a is the loan amount, r is the monthly interest rate, and n is the number of payments

EXERCISE

- Write a program with functions that draws a special version of the Olympic flag for the 2020 games.



EXERCISE

- Develop a module replicating some of the basic functionality of the GoPiGo3 library:
  - Drive meters
  - Turn radians
  - Go to point (relative to current location)
  - Orbit left/right
- Only use methods that engage/disengage motors and the timing library (e.g., you can use forward but not drive_cm)