

File I/O and Subroutines

Michael C. Hackett
Computer Science Department

Community
College
of Philadelphia

Lecture Topics

- Writing data to text files.
- Reading data from text files.
- Appending data to text files.
- Subroutines
- Global and Local Variables
- Parameters and Arguments

Colors/Fonts

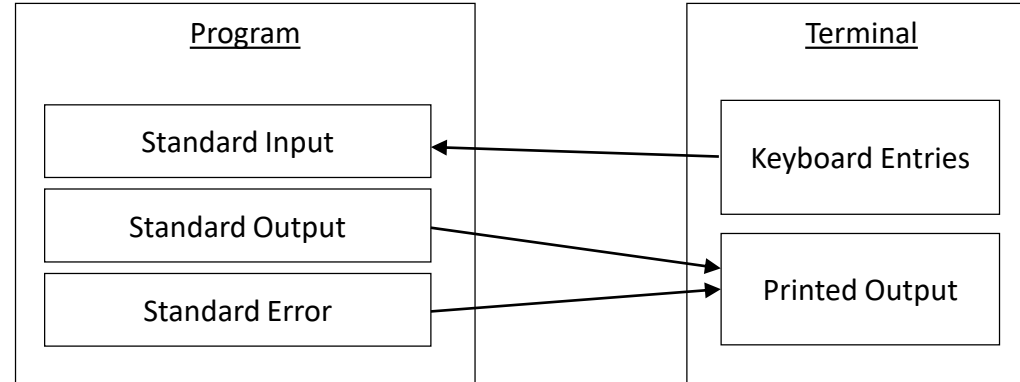
• Global Variable Names	—	Brown
• Local Variable Names	—	Lt Blue
• Literals	—	Blue
• Keywords	—	Orange
• Operators/Punctuation	—	Black
• Functions	—	Purple
• Parameters	—	Gold
• Comments	—	Gray
• Modules	—	Pink

Source Code — **Consolas**
Output — Courier New

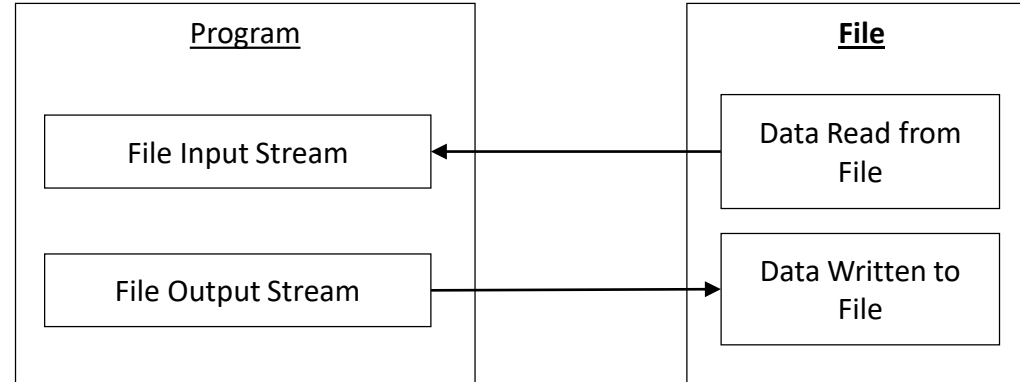
What are files?

- A ***file*** is stream of binary, digital information typically kept on a long-term storage device.
 - Word documents, Powerpoint presentations, and PDFs are all examples of different types of files.
- Can be used as an input data stream. (“Reading a file”)
- Can be used as an output data stream. (“Writing to a file”)

Standard Data Streams (Shown Previously)



File Data Streams



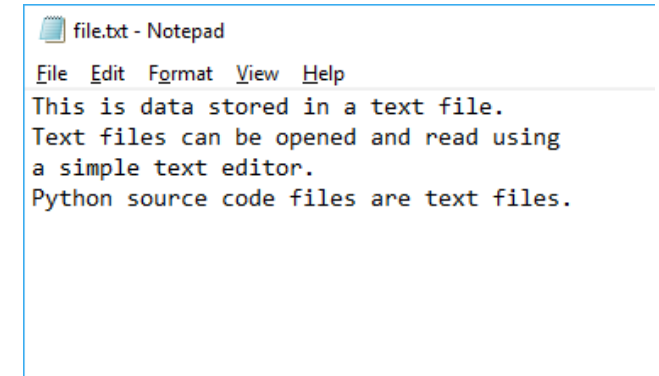
Extensions

- A file has a name which normally includes an extension.
 - Textfile.**txt**
 - WordDocument.**docx**
- You can have files without extensions.
 - Extensions are primarily used by the operating system, so it knows what program to use to open and read the file.
 - Some programs will only accept files with certain extensions.

Types of Files

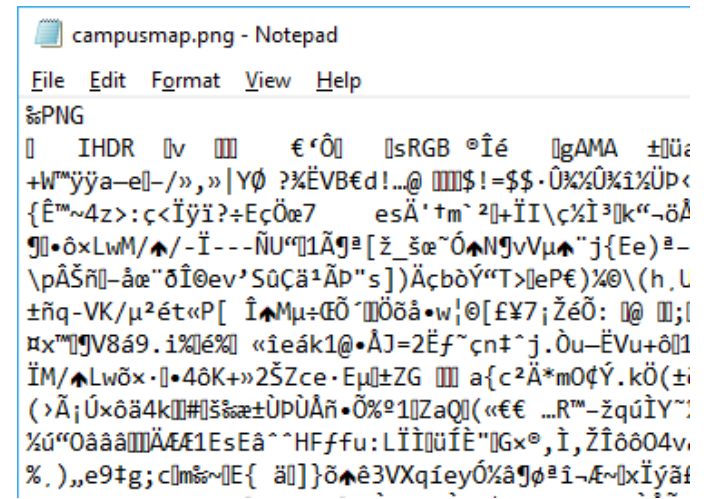
- Text Files

- The binary information contained in the file is encoded with ASCII plaintext.
- Can be opened in any text editor (like Notepad.)
- “Human readable”



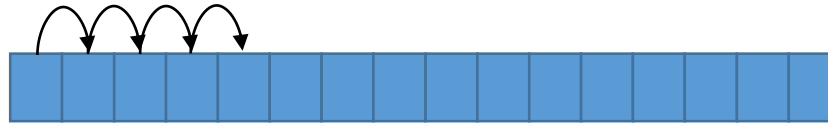
- Data Files

- Files that are not stored in plaintext, like images and compiled programs.
- Normally cannot be opened in any text editor.
- Raw binary- “Computer readable”

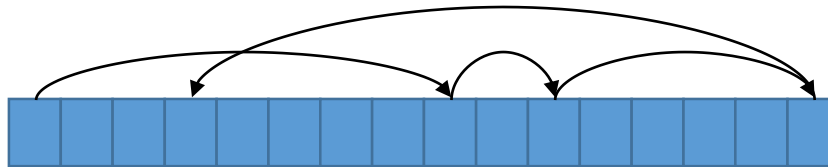


File Access

- Using ***sequential access***, data is read/accessed from the beginning of the file through the end of the file.



- Using ***direct access***, data can be accessed from any location in the file.
 - A topic discussed in CSCI 112



Opening a File

- To open a (text file) data stream in Python, use its built-in **open** function.
- The open function accepts two arguments: The file's name and the mode in which the file is being used.
 - Both arguments are strings.

```
my_text_file = open(filename, mode)
```

- The object returned by the open function is a file object.

Specifying the File's name/path

- If the file you wish to access is in the same folder as the Python program opening the file, you only need to supply the file's name.

```
my_text_file = open("file.txt", mode)
```

- If the file is in a subfolder, you'll need to supply the path to the file beginning with the subfolder's name.

```
my_text_file = open("subfolder\\subfolder2\\file.txt", mode)
```

- Remember, a backslash in a String literal indicates an escape sequence.

Specifying the File's name/path

- If the file is in an entirely different folder, you'll need to supply the full path to the file (beginning with the drive letter on Windows).

```
my_text_file = open("C:\\path\\to\\the\\file.txt", mode)
```

Writing Data to a Text File

- Specifying "w" as the mode will open the file in write mode.

```
my_output_file = open("output.txt", "w")
```

- In write mode, data can be written to the file.
 - If the specified file ***does not*** already exist (you want to make a new file) Python will create it.
 - If the specified file ***does*** exist, its contents will be **ERASED**.

Saving a File

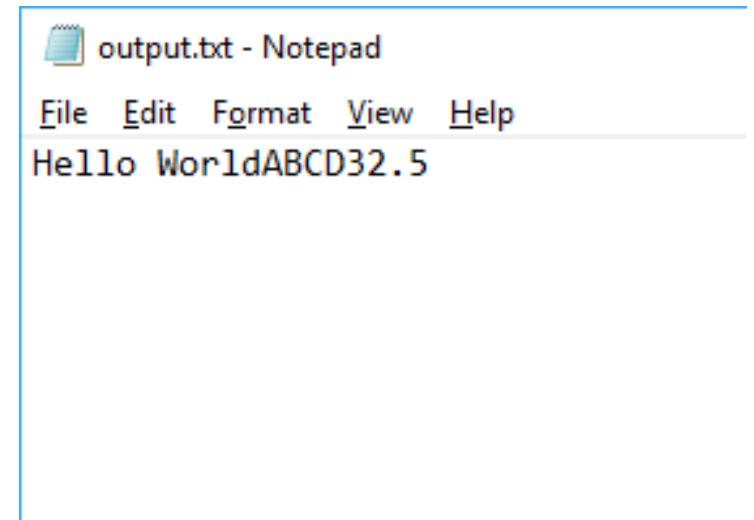
- To when you are finished writing to the output stream, call the **close** function.
- This will save the file.
 - If you do not close the stream, the information you wrote will not be saved.

```
my_output_file.close()
```

Writing Data to a New Text File

- Once the stream is open, we can write data to the file.
- A file's **write** function will write string values to the file.
 - If the data is numeric (ints or floats) be sure to typecast the data to string form.

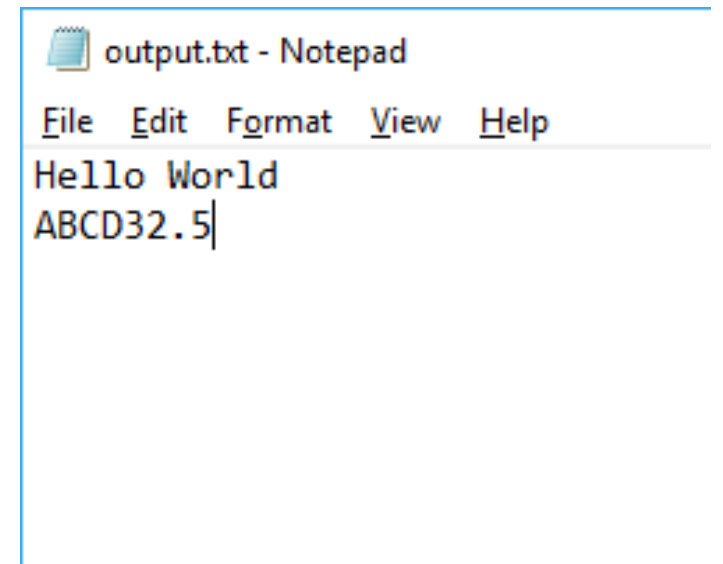
```
my_output_file = open("output.txt", "w")  
my_output_file.write("Hello World")  
my_output_file.write("ABCD")  
my_output_file.write(str(32.5))  
my_output_file.close()
```



Writing Data to a New Text File

- The write function does not add line feeds after each function call.
- To add line feeds, add (or concatenate) `\n` to the end of the line.

```
my_output_file = open("output.txt", "w")  
my_output_file.write("Hello World\n")  
my_output_file.write("ABCD")  
my_output_file.write(str(32.5))  
my_output_file.close()
```



Reading Data from a Text File

- Specifying "r" as the mode will open the stream in read-only mode.

```
my_text_file = open("file.txt", "r")
```

- No data can be written to a file opened in read-only mode.

Closing a File

- To when you are finished reading from the input stream, call the **close** function.
- Python can't have two instances of the same file open.
 - Always close your file when you are done reading from it.

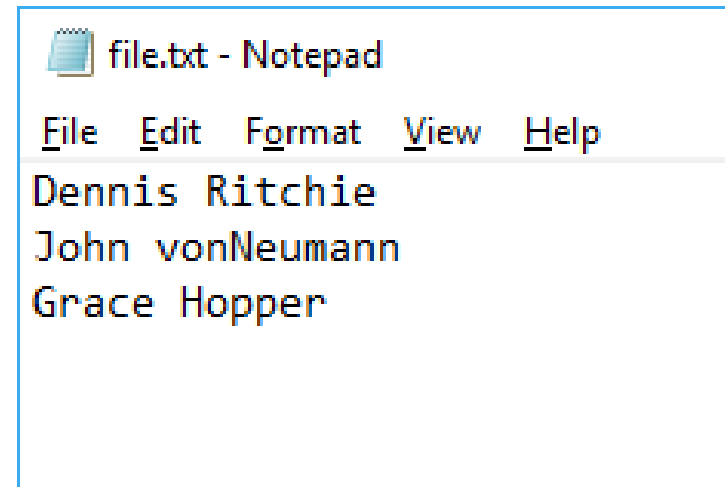
```
my_text_file.close()
```

Reading Data from a Text File

- Once the file is opened in read mode, we can read the contents of the file.
- To read a file, line-by-line, use the file's readline function.
 - The function will return a string containing the next line in the file.

```
my_text_file = open("file.txt", "r")  
line1 = my_text_file.readline()  
print(line1)  
my_text_file.close()
```

Dennis Ritchie



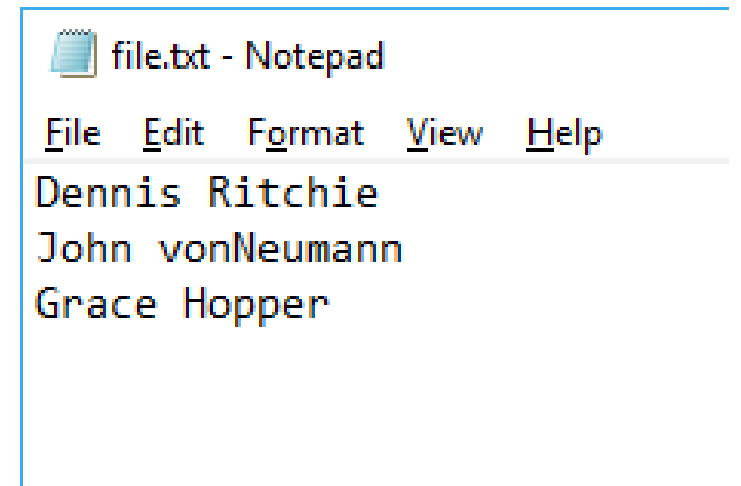
Reading Data from a Text File

```
my_text_file = open("file.txt", "r")
line1 = my_text_file.readline()
line2 = my_text_file.readline()
line3 = my_text_file.readline()
print(line1)
print(line2)
print(line3)
my_text_file.close()
```

Dennis Ritchie

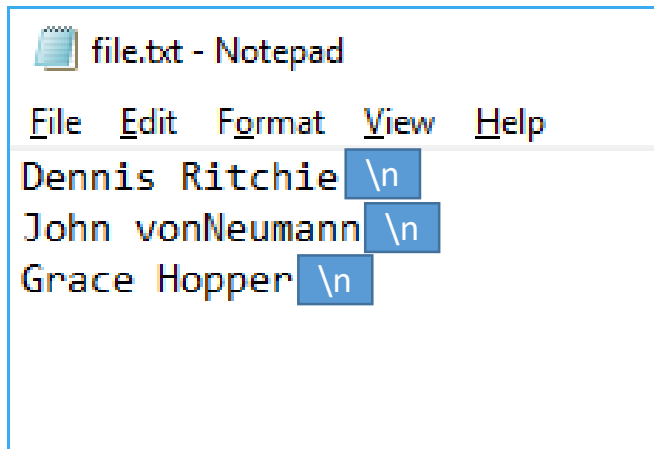
John vonNeumann

Grace Hopper



Reading Data from a Text File

- The extra lines are the result of the non-character line feed (`\n`) at the end of each line in the file.



```
my_text_file = open("file.txt", "r")
line1 = my_text_file.readline()
line2 = my_text_file.readline()
line3 = my_text_file.readline()
print(line1)
print(line2)
print(line3)
my_text_file.close()
```

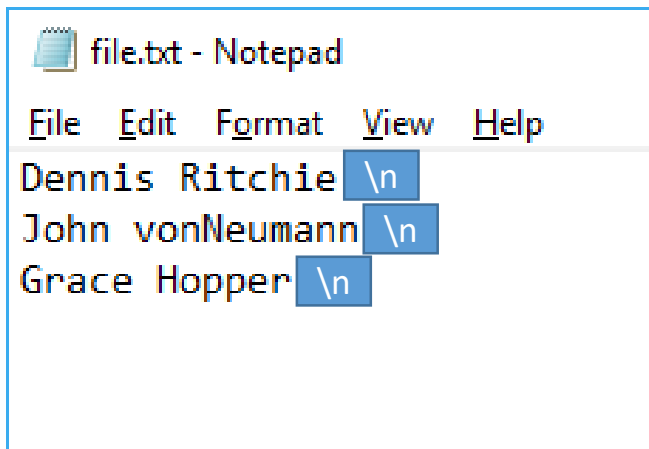
Dennis Ritchie

John vonNeumann

Grace Hopper

Reading Data from a Text File

- To strip away the line feed, we can use the string's **rstrip** function.



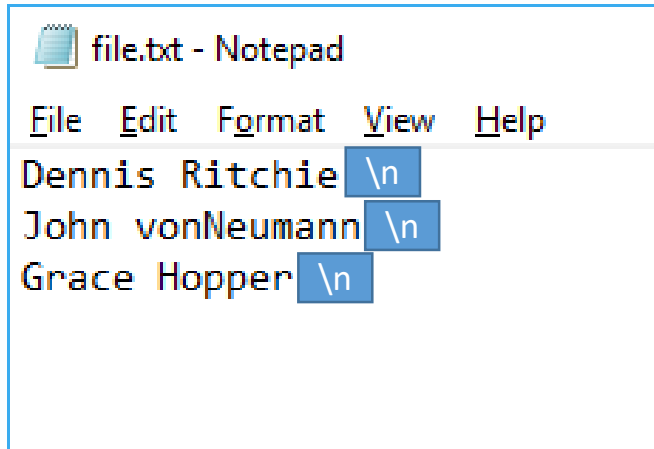
```
file.txt - Notepad
File Edit Format View Help
Dennis Ritchie \n
John vonNeumann \n
Grace Hopper \n
```

```
my_text_file = open("file.txt", "r")
line1 = my_text_file.readline().rstrip("\n")
line2 = my_text_file.readline()
line3 = my_text_file.readline()
print(line1)
print(line2)
print(line3)
my_text_file.close()
```

Dennis Ritchie
John vonNeumann

Grace Hopper

Reading Data from a Text File



```
my_text_file = open("file.txt", "r")
line1 = my_text_file.readline().rstrip("\n")
line2 = my_text_file.readline().rstrip("\n")
line3 = my_text_file.readline().rstrip("\n")
print(line1)
print(line2)
print(line3)
my_text_file.close()
```

```
Dennis Ritchie
John vonNeumann
Grace Hopper
```

Reading Data from a Text File

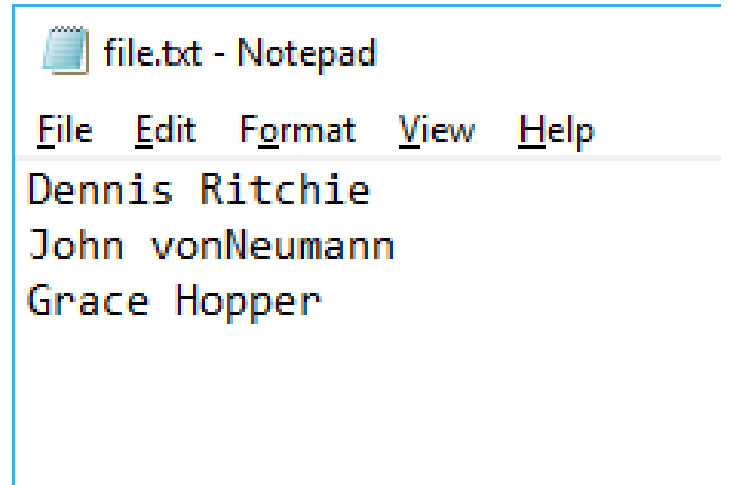
- A for loop can be used to read through a file sequentially.

```
my_text_file = open("file.txt", "r")
```

```
for line in my_text_file :  
    print(line.rstrip("\n"))
```

```
my_text_file.close()
```

```
Dennis Ritchie  
John vonNeumann  
Grace Hopper
```



Appending Data to a Text File

- Specifying "a" as the mode will open an output stream in append mode.

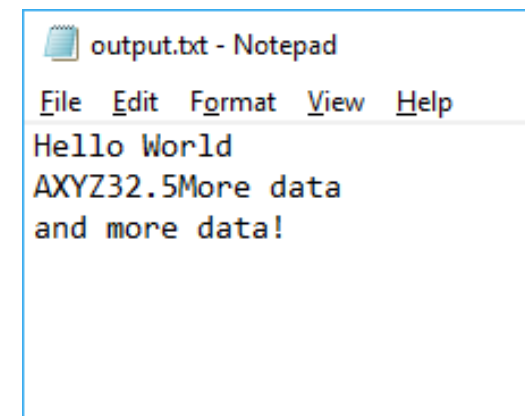
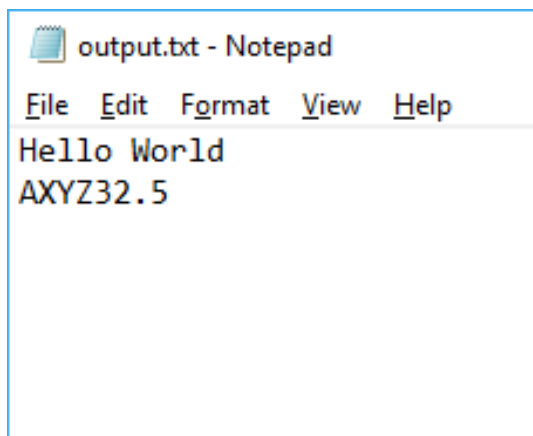
```
my_existing_file = open("output.txt", "a")
```

- In append mode, data can be written to a new or existing file.
 - If the file does not already exist, Python will create it.
 - If the file does exist, the file will be opened and wait for more data to be written to the end of the file.
- Be sure to close the file when you are finished appending to it.

Appending Data to a Text File

- Once the file is open, we can continue writing data to the file.

```
my_output_file = open("output.txt", "a")  
my_output_file.write("More data\n")  
my_output_file.write("and more data!")  
my_output_file.close()
```



Subroutines

- A ***subroutine*** is a group of self-contained instructions that are executed when called.

- A **function** is a subroutine that returns data when called.

```
my_existing_file = input("Enter your name: ")
```

- A **procedure** is a subroutine that does not return data when called.

```
print("Hello World!")
```

- A **method** is a subroutine (function or procedure) that is part of a software object.

```
some_string.replace("Monday", "Tuesday")
```

- All four terms are generally used interchangeably.
 - These are the correct definitions, though.

Creating Subroutines

- Subroutines begin with the keyword **def** (short for define.)
- A parameter list is optional (but the parentheses are not.)
- Subroutine names follow the same rules as variables.

```
def name(parameter list):  
    #code that will be  
    #executed
```

└─┬─┘
Indent one tab.

Creating a Procedure

- In Python, a subroutine must be defined before it can be called.

```
def test_procedure():  
    print("Hello World")
```

```
test_procedure()
```

```
Hello World
```

Procedures

```
def say_goodbye() :      4  
    print("Goodbye World")
```

```
def say_hello() :       2  
    print("Hello World")
```

```
say_hello()    1  
say_goodbye()  3
```

```
Hello World  
Goodbye World
```

Creating a Function

- Functions are like procedures, except they return data when called.
- The return keyword is used to give data back when the function is called.

```
def print_total():  
    total = 5 + 6  
    return total
```

```
value = print_total()  
print(value)
```

11

Procedures and Functions

```
def print_total():          4
    total = 5 + 6
    return total
```

```
def say_hello() :          2
    print("Hello World")
```

```
say_hello()                1
value = print_total()      3
print(value)
```

```
Hello World
11
```


Global Variables

- A ***global variable*** is a variable that exists outside of any one subroutine.
- Global variables are normally declared at the beginning of the source code file.
 - Although, subroutines can access a global variable regardless of where it is declared.

Global Variables

```
global_variable = 3
```

```
def test1() :  
    print("global_variable test1:", global_variable)  
    test2()
```

```
def test2() :  
    print("global_variable test2:", global_variable)
```

```
test1()
```

```
global_variable test1: 3  
global_variable test2: 3
```

Local Variables

- A ***local variable*** is a variable exists only inside of a function.
- Local variables are inaccessible to code outside of the function.
- Where a variable can be used is called its ***scope***.

Local Variables

```
global_variable = 3
```

```
def test() :
```

```
    local_variable = 8.6
```

```
    print("global_variable:", global_variable)
```

```
    print("local_variable:", local_variable)
```

```
test()
```

Scope of global_variable

Scope of local_variable

```
global_variable: 3
```

```
local_variable: 8.6
```

Global Variables Declared Locally

- A local variable can be made global using the **global** keyword.
 - Its declaration and assignment must be on separate lines.


```
global my_variable  
my_variable = 75.4
```

Global Variables Declared Locally

```
global_variable = 3
```

```
def test() :  
    local_variable = 8.6  
    global my_variable  
    my_variable = 75.4  
    print("global_variable:", global_variable)  
    print("local_variable:", local_variable)  
    print("my_variable:", my_variable)
```

```
print("global_variable:", global_variable)  
print("my_variable:", my_variable)  
test()
```



Will not work because its
containing function has
not been called yet.

Global Variables Declared Locally

```
global_variable = 3
```

```
def test() :  
    local_variable = 8.6  
    global my_variable  
    my_variable = 75.4  
    print("global_variable:", global_variable)  
    print("local_variable:", local_variable)  
    print("my_variable:", my_variable)
```

```
test()  
print("global_variable:", global_variable)  
print("my_variable:", my_variable)
```

The diagram consists of five blue arrows pointing from the code to the output. The first arrow points from the `global_variable` argument in the first `print` statement inside the `test` function to the first line of output, `global_variable: 3`. The second arrow points from the `local_variable` argument in the second `print` statement inside the `test` function to the second line of output, `local_variable: 8.6`. The third arrow points from the `my_variable` argument in the third `print` statement inside the `test` function to the third line of output, `my_variable: 75.4`. The fourth arrow points from the `global_variable` argument in the first `print` statement outside the `test` function to the fourth line of output, `global_variable: 3`. The fifth arrow points from the `my_variable` argument in the second `print` statement outside the `test` function to the fifth line of output, `my_variable: 75.4`.

```
global_variable: 3  
local_variable: 8.6  
my_variable: 75.4  
global_variable: 3  
my_variable: 75.4
```

Global Variables Declared Locally

```
global_variable = 3
```

```
def test1() :  
    local_variable = 8.6  
    global my_variable  
    my_variable = 75.4  
    print("global_variable:", global_variable)  
    print("local_variable:", local_variable)  
    print("my_variable:", my_variable)  
    test2()
```

```
def test2() :  
    print("my_variable:", my_variable)
```

```
test1()
```

global_variable: 3
local_variable: 8.6
my_variable: 75.4
my_variable: 75.4

Parameters and Arguments

- Subroutines have the ability to accept arguments.
 - ***Arguments*** are data passed to a function

```
round(original_number)
```

```
format(temp, "f")
```

- The variables used in the source code of a function to represent the arguments are called ***parameters***.

Parameters and Arguments

```
def main() :  
    calculate_area(10, 20)  
  
def calculate_area(length_in, width_in) :  
    area = length_in * width_in  
    print("The area is", area)  
  
main()
```

Arguments – Data passed in

Parameters – Represent the arguments passed

The area is 200

Parameters and Arguments

```
def main() :  
    length = 10  
    width = 20  
    calculate_area(length, width)  
  
def calculate_area(length_in, width_in) :  
    area = length_in * width_in  
    print("The area is", area)  
  
main()
```

Arguments – Data passed in


Parameters – Represent the arguments passed in

The area is 200

Parameters and Arguments

- The number of arguments must match the number of parameters.

```
def main() :  
    calculate_area(10)  
  
def calculate_area(length_in, width_in) :  
    area = length_in * width_in  
    print("The area is", area)  
  
main()
```

 Error

Parameters and Arguments

```
def main() :  
    area = calculate_area(10, 20)  
    print("The area is" + str(area))
```

```
def calculate_area(length_in, width_in) :  
    area = length_in * width_in  
    return area
```

```
main()
```

```
The area is 200
```


Note: The main procedure's local variable `area` and the `calculate_area` function's local variable `area` are two **different** variables.

They can have the same name because they are local to their respective function.

Parameters and Arguments

- Functions can have more than one return statement.

```
def main() :  
    eligible = can_vote(22)  
    print("Eligible to vote:", ("Yes" if eligible else "No"))
```



In-line If Statement

```
def can_vote(age_in) :  
    if age_in >= 18 :  
        return True  
    else :  
        return False
```

Eligible to vote: Yes

```
main()
```