Basic Python II

K.N. Toosi

Fall 2022

Functions

Built-in Functions

- Examples:
  - My car has a built-in radio. The radio is not really the car itself; it was developed separately, probably by a different company. But when I bought the car, it had a radio in it. When I want the radio to do something, I press its buttons or turn its dials, and the radio responds appropriately. **I don't need to know how the radio works**, **I just need to know how to use the radio's controls**.
  - Similarly, in a typical kitchen, there are a number of built-in appliances. The inner workings of a home bread maker are a mystery to most people, but the average person can read the manual and figure out how to use the controls to have it make a delicious loaf of bread.
- Python has a number of things like this called built-in functions. They are pieces of code that are available for you to use in your programs. When you want to use a built-in function, you specify its name and typically give it some information. The function does some work with that information and gives you back some results.

Definitions

- **Function Call** Using a function is known as *calling a function*, or making a function call, or making a call to a function. To call a function, you supply the name of the function, followed by a set of open/close parentheses. This is the generic form:

  <functionName>()

- **Data passing** Most built-in functions expect you to supply one or more pieces of information in addition to the function name. This is commonly called *passing data to a function*.

- **Arguments** An argument is a value that is passed when you call a function. Inside the function call's parentheses, you put any data you want, called an argument, to send to that function. Here's what a generic call to a function with arguments looks like:

  <functionName>(<argument1>, <argument2>, ...)

Definitions

- **Results** When you call a built-in function, the function does its work and typically hands back a result. When the function is finished, the result replaces the call and its arguments; that is, the line continues to execute using the returned value in place of the call. Very often when you make a call to a function, you assign the returned value to a variable using an assignment statement, as follows:

  <variable> = <functionName>(<argument1>, <argument2>, ...)

Trivial examples

- type( )
- input( )
- Conversion Functions:

  Python has three built-in conversion functions that can change a value from one data type to another: the int function, the float function, and the str function.

Definitions – math module

- A **module** is a file that contains a collection of related functions.
- Before we can use the functions in a module, we have to import it with an import statement:

  >>> import math

- This statement creates a module object named math. The module object contains the functions and variables defined in the module.
- To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot. This format is called dot notation.

Math module

- Example:

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

- Composition

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
x = math.exp(math.log(x+1))
```

User-defined Function

- **Function**: A function is a series of related steps (statements) that make up a larger task, which is often called from multiple places in a program.
- Here is the generic form of a function in Python:

```
def <functionName>(<optionalParameters>):
    <indented statement(s)> # the 'body' of the function
```

- The word def is short for definition. You are defining a function. def is one of the special reserved Python keywords.
- Calling a User-Defined Function:

```
<functionName>(<argument1>, <argument2>, ...)
```

- Calling the function, must have parentheses, even if there are no arguments

Simple Example

```
# This function just prints a line of asterisks followed by a blank line
def separateRuns():
        print('**********')
        print() #blank line
def getGroceries():
        print('milk')
        print('flour')
        print('sugar')
        print('butter')
        separateRuns() # call another function
# Main code starts here:
getGroceries()
getGroceries()
```

- what happens when you run this code?
  - Python sees the first def statement for separateRuns and recognizes that it is the definition of a function. Python remembers where this function is and skips over the body of the function (the indented lines).
  - Next, it sees the def statement for the getGroceries function. Again, it remembers where this is and skips over the body of this function.
  - Eventually, it finds the first real line of the executable code—the first call to getGroceries().
  - When this line runs, Python remembers where it was, and execution jumps to the def statement for the getGroceries function. At the last statement, Python finds a call to another function: separateRuns. Python remembers where it came from and transfers control into that function.
  - The Control is transferred back to where it was called from (inside getGroceries). And because this is the last line of getGroceries, control is transferred back to where that function was called from (in the main code).
  - On the next line, Python finds another call to getGroceries. Control is transferred into the function once again, and the entire sequence is repeated. After the second call to getGroceries completes, Python finds no more lines of code to run, and the program terminates.

-cnt-

- Now, let's modify getGroceries again, this time so that it accepts four parameters. We'll also change each hard-coded print statement to print an appropriate parameter:

```
def getGroceries(item1, item2, item3,
item4):
    print(item1)
    print(item2)
    print(item3)
    print(item4)
```

- >>> getGroceries('eggs', 'soap', 'lettuce', 'cat food')

- The variables items1, items2,… in the definition of getGroceries( ) are examples of **parameters**, piece of information the function needs to do its job.
- The values 'eggs', 'soap', 'lettuce',… in getGroceries('eggs', 'soap', 'lettuce', 'cat food') are examples of **arguments**. An argument is a piece of information that's passed from a function call to a function.

Passing Arguments

Positional argument

- When you call a function, Python must match each argument in the function call with a parameter in the function definition. The simplest way to do this is based on the order of the arguments provided. Values matched up this way are called positional arguments.

```
def describe_pet(animal_type, pet_name):
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is
    {pet_name.title()}.")
describe_pet('hamster', 'harry')
```

Order Matters in Positional Arguments

Keyword Arguments

- A keyword argument is a name-value pair that you pass to a function. You directly associate the name and the value within the argument.

```
describe_pet(animal_type='hamster', pet_name='harry')
```

- The order of keyword arguments doesn't matter because Python knows where each value should go. When you use keyword arguments, be sure to use the exact names of the parameters in the function's definition.

Default Values

- When writing a function, you can define a default value for each parameter. If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter's default value.

```
def describe_pet(pet_name, animal_type='dog'):
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")
describe_pet(pet_name='willie')
```

- **Note:** When you use default values, any parameter with a default value needs to be listed after all the parameters that don't have default values. This allows Python to continue interpreting positional arguments correctly.

positional argument **SHOULD NOT** follow keyword argument

Wrong

```
def team(name, project, members="Ali, Mohammad"):
    print(name, "with members", members,"is working on an", project)

team(name = "KNTU", "LRMA")
```

Correct

```
def team(name, project, members="Ali, Mohammad"):
    print(name, "with members",members,"is working on an", project)

team("KNTU", project = "LRMA")
```

Passing a List

- When you pass a list to a function, the function can modify the list. Any changes made to the list inside the function's body are permanent, allowing you to work efficiently even when you're dealing with large amounts of data.
- Sometimes you'll want to prevent a function from modifying a list.
- You can send a copy of a list to a function like this:

  function_name(list_name[:])

- Even though you can preserve the contents of a list by passing a copy of it to your functions, you should pass the original list to functions unless you have a specific reason to pass a copy. It's more efficient for a function to work with an existing list to avoid using the time and memory needed to make a separate copy, especially when you're working with large lists.

Passing an Arbitrary Number of Arguments

- Sometimes you won't know ahead of time how many arguments a function needs to accept. Fortunately, Python allows a function to collect an arbitrary number of arguments from the calling statement.

```
def make_pizza(*toppings):
        print(toppings)
make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

- The asterisk in the parameter name *toppings tells Python to make an empty tuple called toppings and pack whatever values it receives into this tuple. The print() call in the function body produces output showing that Python can handle a function call with one value and a call with three values. It treats the different calls similarly. Note that Python packs the arguments into a tuple, even if the function receives only one value:

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

Example

```
# program to find sum of multiple numbers

def find_sum(*numbers):
    result = 0

    for num in numbers:
        result = result + num

    print("Sum = ", result)

# function call with 3 arguments
find_sum(1, 2, 3)

# function call with 2 arguments
find_sum(4, 9)
```

Avoiding Argument Errors

- When you start to use functions, don't be surprised if you encounter errors about unmatched arguments. Unmatched arguments occur when you provide fewer or more arguments than a function needs to do its work.

  Traceback (most recent call last):

  File "pets.py", line 6, in <module>

  describe_pet()

  TypeError: describe_pet() missing 2 required positional arguments: 'animal_type' and 'pet_name'

- If you provide too many arguments, you should get a similar traceback that can help you correctly match your function call to the function definition.

The return Statement

Returning No Value: None

- When a function does not have an explicit return statement, Python builds an implied return statement that returns no value. In fact, you can write a return statement that does not give back a value:

```
return # no return value specified
```

- But when a return statement is executed without any returned value, Python actually returns a special value of **None**. None is a Python keyword that means no value.
- A function that doesn't return a value is called a **void function**.
- The value None is not the same as the string 'None'. It is a special value that has its own type:

```
>>> print(type(None))
<class 'NoneType'>
```

Returning a Simple Value

- Example:

```
def square(number):
    answer = number * number
    return answer
sqrt = square(10)
```

-cnt-

Returning More Than One Value

- Python has a further extension of the return statement. In most other programming languages, the return statement can only return either no values or a single value. In Python, just as you can pass as many values as you want into a function, you can also return any number of values:

  return <value1>, <value2>, <value3>, ...

- For example, you could create a function that returns three values, like this:

```
def myFunction(parameter1, parameter2):
    # Body of the function
    return answer1, answer2, answer3
```

- Then you would call myFunction with code like this:

```
variable1, variable2, variable3 = myFunction(argument1, argument2)
```

- The order of the variables in the assignment statement matches the order of the answer variables in the function's return statement.

Scope

- Variables have a lifetime.
- **Scope** is the amount of code over which a variable is active.
- Two levels of scope: **global and local**.

- **Global variables are created at the top level of a program, in the main code**. They have what is called global scope. Global variables maintain their values and are available throughout a program.
- Global variables (created in the main program code) can legally be used inside functions. However, it is strongly recommended to never do this. Using global variables inside functions leads to a poor coding practice called **spaghetti code**.
- Instead of using global variables inside functions, **whenever a function needs a value that is held in a global variable, that value should be passed as an argument** into the function when the function is called. **If the function wants to effectively change the value of a global variable, the function should return a value**, and the caller can set a new value for the global variable as the result of the call.

Scope – cnt -

- **Local** variables are created inside a function.
- The scope of a local variable ranges from the point where it is first used in a function to the end of that function.
- When the function ends, any local variables used inside the function literally disappear.
- You cannot access local variables outside of a function because those variables are out of scope and no longer exist.

Scope –cnt-

```
def myFunction():
    someVariable = 5

someVariable = 10
myFunction()
print(someVariable)
```

```
def myFunction():
    global someVariable
    someVariable = someVariable + 1

someVariable = 20
myFunction()
print(someVariable)
```

```
def myFunction(aVariable):
    aVariable = aVariable + 1
    return aVariable

someVariable = 20
someVariable = myFunction(someVariable)
print(someVariable)
```

Hands on code! Example 1

- Write a Python function to find the Max of three numbers.

```
def max_of_two( x, y ):
    if x > y:
        return x
    return y
def max_of_three( x, y, z ):
    return max_of_two( x, max_of_two( y, z ) )
print(max_of_three(3, 6, -5))
```

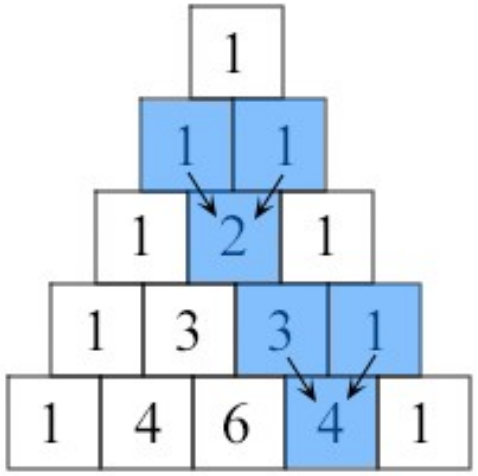
Example 2

- Write a Python program to reverse a string.
  - Sample String: “1234abcd”
  - Expected Output: “dcba4321”

```
def string_reverse(str1):
    rstr1 = ''
    index = len(str1)
    while index > 0:
        rstr1 += str1[ index - 1 ]
        index = index - 1
    return rstr1
print(string_reverse('1234abcd'))
```

Example 3

- Write a Python function that prints out the first n rows of Pascal's triangle.




```
def pascal_triangle(n):
  trow = [1]
  y = [0]
  for x in range(max(n,0)):
    print(trow)
    #trow=[l+r for l,r in zip(trow+y, y+trow)]
    trow1 = trow + y
    trow2 = y + trow
    trow.clear()
    for i in range(len(trow1)):
        trow.append(trow1[i]+trow2[i])
  return n>=1


pascal_triangle(6)
```

Why Functions?

- Creating a new function gives you an opportunity to name a group of statements, which makes your program **easier to read**.
- Functions can **make a program smaller** by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to **debug the parts one at a time** and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can **reuse** it.

Storing Your Functions in Modules

- One advantage of functions is the way they **separate blocks of code** from your main program. By using descriptive names for your functions, your main program will be much easier to follow.
- You can go a step further by **storing your functions in a separate file** called a **module** and then **importing** that module into your main program.
- An import statement tells Python to make the code in a module available in the currently running program file.
- Storing your functions in a separate file allows you to
  - **hide the details** of your program's code and focus on its higher-level logic.
  - It also allows you to **reuse functions** in many different programs. When you store your functions in separate files, you can share those files with other programmers without having to share your entire program.

Import a module

- Importing an **Entire** Module:
  - This first approach to importing, in which you simply write import followed by the name of the module, makes every function from the module available in your program. If you use this kind of import statement to import an entire module named module_name.py, each function in the module is available through the following syntax:

    module_name.function_name()

- Importing **Specific** Functions:
  - You can also import a specific function from a module. Here's the general syntax for this approach:

    From module_name import function_name
  - With this syntax, you don't need to use the dot notation when you call a function. We can call it by name when we use the function.

Importing: different ways

Definitions

1. from *module_name* import *function_name*
2. from *module_name* import *function_name* as *fn*
3. import *module_name* as *mn*
4. from *module_name* import *

Examples

1. from math import sin

   print(sin(3.14/6))
2. from math import sin as sn

   print(sn(3.14/6))
3. import math as mth

   print(mth.sin(3.14/6))
4. from math import *

   print(sin(3.14/6))

__name__ & __main__

- __name__ variable helps you to check **if the file is being run directly or if it has been imported**.
- if __name__ == '__main__':
  - the code within the 'if' block will be executed only when the code runs directly. Here 'directly' means 'not imported'.
- When Python runs the "source file" as the main program, it sets the special variable (__name__) to have a value ("__main__").

What are the outputs of these scripts? Why different?

1.
```
def main():
    print ("Hello World!")
print ("KNTU FP")
```

2.
```
def main():
    print("Hello World!")
if __name__ == "__main__":
    main()
print("KNTU FP")
```

__name__ & __main__

test.py

```
print("Good Morning")
print("Value of implicit variable __name__ is: ", __name__)
def main():
        print("Hello Python")
print("Good Evening")
if __name__ == "__main__":
        main()
```

test1.py

```
import test
print("Hello World")
```

Recursion

- It is legal for one function to call another; it is also legal for **a function to call itself**. For example, look at the following function:

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

What happens?

- >>> countdown(3)

  The execution of countdown begins with n=3, and since n is greater than 0, it outputs the value 3, and then calls itself...

  > The execution of countdown begins with n=2, and since n is greater than 0, it outputs the value 2, and then calls itself...
  >
  > > The execution of countdown begins with n=1, and since n is greater than 0, it outputs the value 1, and then calls itself...
  > >
  > > > The execution of countdown begins with n=0, and since n is not greater than 0, it outputs the word, "Blastoff!" and then returns.
  > >
  > > The countdown that got n=1 returns.
  >
  > The countdown that got n=2 returns.

  The countdown that got n=3 returns.

- And then you're back in __main__. So, the total output looks like this:

  3

  2

  1

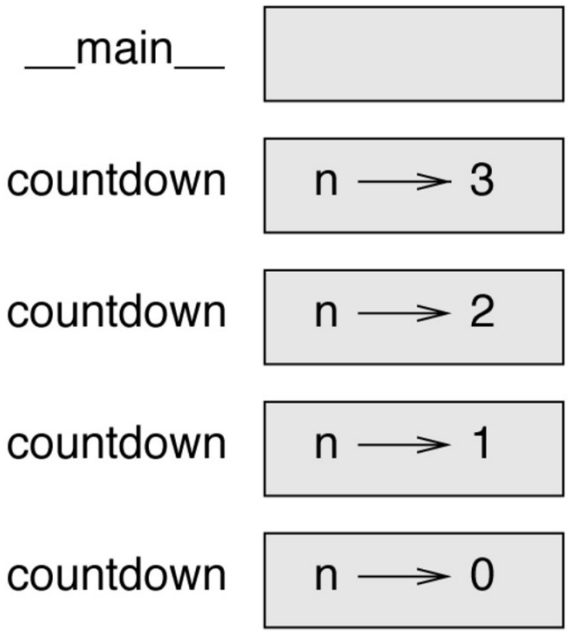
  Blastoff!

- Need for Recursion:

  For simple examples like this, it is probably easier to use a for loop. But we will see examples later that are hard to write with a for loop and easy to write with recursion, so it is good to start early.

- Properties of Recursion:
  - Performing the same operations multiple times with different inputs.
  - In every step, we try **smaller** inputs to make the problem smaller.
  - **Base condition is needed** to stop the recursion otherwise infinite loop will occur.

Stack Diagrams for Recursive Functions



Infinite Recursion – stack overflow

- If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as infinite recursion, and it is generally not a good idea. Here is a minimal program with an infinite recursion:

```
def recurse():
    recurse()
```

- In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:
- How to avoid:
  - If you write an infinite recursion by accident, **review your function to confirm that there is a base case that does not make a recursive call**. And **if there is a base case, check whether you are guaranteed to reach it**.

A Mathematical Interpretation

- Let us consider a problem that a programmer has to determine the sum of first n natural numbers:
  - **approach(1)** – Simply adding one by one

    $$f(n) = 1 + 2 + 3 + \ldots\ldots + n$$

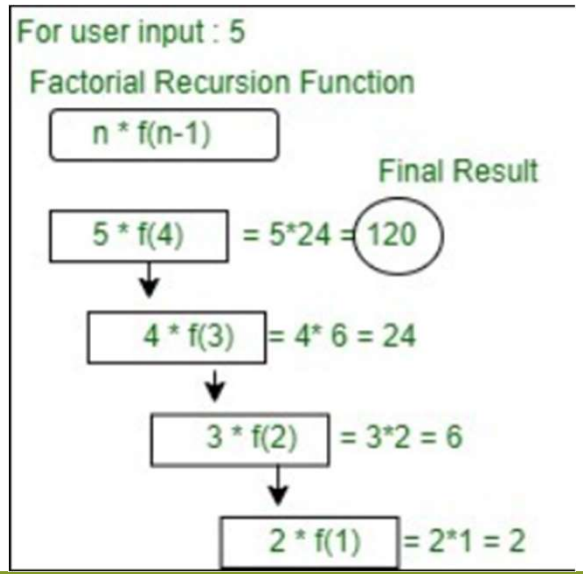
  - **approach(2)** – Recursive adding

    $$f(n) = 1 \qquad n=1$$

    $$f(n) = n + f(n-1) \qquad n>1$$

Example

- Write a program and recurrence relation to find the Factorial of n where n>2 .
- Mathematical Equation:
  - 1 if n = 0 or n = 1;
  - f(n) = n*f(n-1) if n> 1;




```
def f(n):
    # base condition
    if (n == 0 or n == 1):
        return 1

    # Recursive condition
    else:
        return n * f(n - 1)

n = 5
print("factorial of",n,"is:",f(n))
```

Files

Why?

- In every program we have talked about so far, when the program ends, the **computer forgets everything** that happened in the program. Every variable you created, every string you used, every Boolean, every list—it's all gone.
- But what if you want to keep some of the data you generate in a program and save it for when you run the program later? Or maybe you want to save some data so that a different program could use the data you generated.
- If you want to **keep some information around between runs of a program**, you need a way of having what is called **persistent data**—you want the data to remain available on the computer. To do that, you need the ability to **write to and read from a file** on the computer.

Defining a Path to a File

- When you want to read or write a text file, you must first **identify which file** you want to write to or read from.
- Definition: A path is a string that uniquely identifies a file on a computer.
- A path is a **string**. There are two different ways to specify a path: **absolute** and **relative**.
- An absolute path is one that starts at the top of the file system on your computer and ends in the name of the file. For example, an absolute path might look like this in Windows:

  C:\MyFolder\MySubFolder\MySubSubFolder\MyFile.txt

- But a path on one computer may not match a path on another one, most code that uses absolute paths is not **portable**.
- A relative path starts in the folder that contains your Python source file. We say that the path is relative to the location of the source file. That means any file we want to use or create resides either in the same folder as your Python source file or in a folder somewhere below that folder.

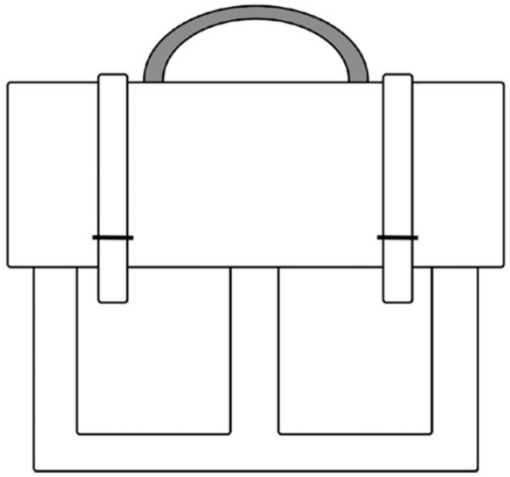
Example of relative path

Folder structure	Path to the text file
PythonSource.py	
MyDataFile.txt	'MyDataFile.txt'
MyFolder	
SomeDataFile.txt	'MyFolder/SomeDataFile.txt'
MySubFolder	
OtherDataFile.txt	'MyFolder/MySubFolder/OtherDataFile.txt'
MySubSubFolder	

Reading & Writing to a file

- Reading text from a file requires three steps:
  1. Open the file for reading
  2. Read from the file (usually into a string variable)
  3. Close the file
- Writing text to a file requires three similar steps:
  1. Open the file for writing
  2. Write a string (usually from a string variable) to the file
  3. Close the file

- Notice that whenever you deal with a file, **you first need to open the file**. In all operating systems, when a program opens a file, the operating system gives back a file handle.



Basic reading & writing in Python

Reading

```
fileHandle = open(filePath, 'r') # r for reading
data = fileHandle.read( ) # read into a variable
fileHandle.close( )
# text read in is now in the variable called data
```

Writing

```
# text to be written is contained in the variable textToWrite
fileHandle = open(filePath, 'w') # w for writing
fileHandle.write(textToWrite) # write out text from a variable
fileHandle.close( )
```

File Access Mode

There are 6 access modes in python.

1. **Read Only ('r') :** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises the I/O error. This is also the default mode in which a file is opened.
2. **Read and Write ('r+'):** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exist.
3. **Write Only ('w') :** Open the file for writing. For the existing files, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exist.
4. **Write and Read ('w+')** : Open the file for reading and writing. For an existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
5. **Append Only ('a')**: Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
6. **Append and Read ('a+') :** Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

File Access Mode

	r	r+	w	w+	a	a+
read	*	*		*		*
write		*	*	*	*	*
create			*	*	*	*
truncate			*	*		
position at start	*	*	*	*		
position at end					*	*

Does the file exist?

- **You cannot read from a file that doesn't exist**. So, you need to check that the file you want to read from actually exists before you attempt to read from it.
- The Python os Package:

```
import os
exists = os.path.exists(filePath)
```

Reading

read( )

- fileHandle.read( ): reads the WHOLE text in the file

readline( )

- fileHandle.readline( ): reads the text line by line

```
file1 = open('myfile.txt', 'r')
count = 0
while True:
    count += 1
    # Get next line from file
    line = file1.readline()
    # if line is empty, end of file is reached
    if not line:
        break
    print("Line{}: {}".format(count, line.strip()))
file1.close()
```

readlines( )

- fileHandle.readlines( ): readlines() is used to read all the lines at a single go and then return them as each line a string element in a list.

```
file1 = open('myfile.txt', 'r')
Lines = file1.readlines()
count = 0
# Strips the newline character
for line in Lines:
    count += 1
    print("Line{}: {}".format(count, line.strip()))
```

Writing

write( )

- fileHandle.write(string): writes the string in the file

writelines( )

- fileHandle.writelines(list): writes the entries of a list in the file

write( ) but append

- fileHandle.write(string): needs to access file with permission to append

  fileHandle = open(filePath, 'a')

Auto close using ‘with’

- Another way to close a file is by using the with clause. It ensures that the file gets closed when the block inside the with clause executes. The beauty of this method is that it doesn’t require to call the close() method explicitly.

```
with open('app.log') as f:
    #do any file operation.
```

Examples

Read/Write to a File in Python

```
with open('myfile.txt', 'w') as f:
   #first line
   f.write('my first file\n')
   #second line
   f.write('This file\n')
   #third line
   f.write('contains three lines\n')


with open('myfile.txt', 'r') as f:
   content = f.readlines()
for line in content:
   print(line)
```

Read from a File in Python

```
with open('myfile.txt', 'w') as f:
   f.write('my first file\n')
   f.write('This file\n')
   f.write('contains three lines\n')
f = open('myfile.txt', 'r')
print(f.read(10))    # read the first 10 data
#'my first f'
print(f.read(4))    # read the next 4 data
#'ile\n'
print(f.read())     # read in the rest till end of file
#'This file\ncontains three lines\n'
print(f.read())  # further reading returns empty sting
#''
```

Example - write into file at position

```
mystring = 'some string'
pos = 10

with open('myfile.txt', 'r+') as f:
    contents = f.read()
    contents = contents[:pos] + mystring + contents[pos + 1:]
    f.seek(0)
    f.truncate()
    f.write(contents)
```

Storing data

- Whatever the focus of your program is, you'll store the information provided in data structures such as lists and dictionaries. You might want to save the information after the program terminates. A simple way to do this involves storing your data using the **json** module.
- The **json module allows you to dump simple Python data structures into a file and load the data from that file the next time the program runs**. You can also use json to share data between different Python programs.

Using json.dump( ) and json.load( )

Let's write a short program that stores a set of numbers and another program that reads these numbers back into memory. The first program will use json.dump( ) to store the set of numbers, and the second program will use json.load( ).

```
import json

numbers = [2, 3, 5, 7, 11, 13]
filename = 'numbers.json'
fileHandle = open(filename, 'w')
json.dump(numbers, fileHandle)
```

==========

```
import json

filename = 'numbers.json'
fileHandle = open(filename, 'r')
numbers = json.load(fileHandle)
print(numbers)
```

Examples - Write a python program to find the longest words.

```
def longest_word(filename):
    with open(filename, 'r') as infile:
            words = infile.read().split()
    word = max(words, key=len)
    return word


print(longest_word('test.txt'))
```

Examples - Write a Python program to count the number of lines in a text file.

```
def file_lengthy(fname):
        with open(fname) as f:
                for i, l in enumerate(f):
                        pass
        return i + 1
print("Number of lines in the file: ",file_lengthy("test.txt"))
```

Examples - Write a Python program to count the frequency of words in a file.

```
def word_count(fname):
    words = {}
    with open(fname) as f:
        text = f.read().lower().split()
        for word in set(text):
            words[word]=text.count(word)
        return words

print("Number of words in the file :",word_count("myfile.txt"))
```

Examples - Write a Python program to combine each line from first file with the corresponding line in second file.

```
with open('file1.txt') as fh1, open('file2.txt') as fh2:
    for line1 in fh1:
        line2 = fh2.readline()
        print(line2+line1)
```

Examples – write multiple objects in Json

```
import json

numbers1 = [1,2,3,4,5,6]
numbers2 = numbers1.copy()
numbers2.reverse()
filename = 'numbers.json'
json_data = {
    'num1': numbers1,
    'num2': numbers2,
}
with open(filename, 'w') as fp:
    json.dump(json_data, fp, indent=4)

with open(filename, 'r') as fp:
    data =json.load(fp)
    print(data['num1'])
    print(data['num2'])
```