

Files and Exceptions

1. Introduction to File Input and Output

1. When a program needs to save data for later use, it writes the data in a *file*. The data can be read from the file at a later time.

2. The programs you have written so far require the user to reenter data each time the program runs, because data that is stored in **RAM** (referenced by variables) disappears once the program stops running.

3. Data is saved in a file, which is usually stored on a computer's disk. Once the data is saved in a file, it will remain there after the program stops running. Data that is stored in a file can be retrieved and used at a later time.

Writing data to a file

Programmers usually refer to the process of saving data in a file as “writing data to the file. When a piece of data is written to a file, it is copied from a variable in RAM to the file (*output file*).

Reading data from a file

The process of retrieving data from a file is known as “reading data from” the file. When a piece of data is read from a file (*input file*), it is copied from the file into RAM, and referenced by a variable.

There are three steps that must be taken when a file is used by a program.

1. *Open the file* - Create (*output file*) or open (*input file*) the file.
2. *Process the file* - Write to the file (*output file*) or read from the file (*input file*)
3. *Close the file* - Closing a file disconnects the file from the program.

Types of Files

A *text file* contains data that has been encoded as text, using a scheme such as ASCII or Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters.

A *binary file* contains data that has not been converted to text. The data that is stored in a binary file is intended only for a program to read.

File Access Methods

Most programming languages provide two different ways to access data stored in a file: *sequential access* and *direct access*.

When you work with a sequential access file, you access data from the beginning of the file to the end of the file.

When you work with a direct access file (or random access file), you can jump directly to any piece of data in the file without reading the data that comes before it.

Filename and File Objects

Filename extensions, which are short sequences of characters that appear at the end of a filename preceded by a period. The extension usually indicates the type of data stored in the file. For example, the **.txt** extension usually indicates that the file contains text.

In order for a program to work with a file on the computer's disk, the program must create a file object in memory. A **file object** is an object that is associated with a specific file, and provides a way for the program to work with that file.

In the program, a variable references the file object. This variable is used to carry out any operations that are performed on the file.

Opening a File

The open function creates a file object and associates it with a file on the disk.

General format: `file_variable = open(filename, mode)`

In the general format:

- *file_variable* is the name of the variable that will reference the file object.
- *filename* is a string specifying the name of the file.
- *mode* is a string specifying the mode (reading, writing, etc.) in which the file will be opened.

Some of the Python file modes:

Mode	Description
'r'	Open a file for reading only. The file cannot be changed or written to.
'w'	Open a file for writing. If the file already exists, erase its contents. If it does not exist, create it.
'a'	Open a file to be written to. All data written to the file will be appended to its end. If the file does not exist, create it.

WARNING:

Remember, when you use the 'w' mode you are creating the file on the disk. If a file with the specified name already exists when the file is opened, the contents of the existing file will be erased.

Specifying the Location of a File

When you pass a filename that does not contain a path as an argument to the open function, the Python interpreter assumes that the file's location is the same as that of the program.

If you want to open a file in a different location, you can specify a path as well as a file name in the argument that you pass to the open function. If you specify a path in a string literal, be sure to prefix the string with the letter r.

Here is an example:

```
test_file = open(r'C:\Users\Blake\temp\test.txt', 'w')
```

The r prefix specifies that the string is a *raw string*. This causes the Python interpreter to read the backslash characters as literal backslashes. Without the r prefix, the interpreter would assume that the backslash characters were part of escape sequences, and an error would occur.

Writing Data to a File

A **method** is a function that belongs to an object, and performs some operation using that object.

Once you have opened a file, you use the file object's methods to perform operations on the file. For example, file objects have a method named **write** that can be used to write data to a file. General format: `file_variable.write(string)`

In the format, *file_variable* is a variable that references a file object, and *string* is a string that will be written to the file. The file must be opened for writing (using the 'w' or 'a' mode) or an error will occur.

Once a program is finished working with a file, it should close the file. Closing a file disconnects the program from the file. In some systems, failure to close an output file can cause a loss of data. In Python you use the file object's **close** method to close a file.

```
file_variable.close()
```

Python program that opens an output file, writes data to it, and then closes it - file_write.py

```
def main():
```

```
    outfile = open('grandmasters.txt', 'w')
```

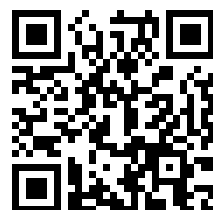
```
    outfile.write('Magnus Carlsen\n')
```

```
    outfile.write('Garry Kasparov\n')
```

```
    outfile.write('Viswanathan Anand\n')
```

```
    outfile.close()
```

```
main()
```



Reading Data From a File

If a file has been opened for reading (using the 'r' mode) you can use the file object's **read** method to read its entire contents into memory. When you call the read method, it returns the file's contents as a string.

Python program that opens an input file, reads data from it, and then closes it -file_read.py

def main():

infile = open('grandmasters.txt', 'r')

file_contents = infile.read()

infile.close()

print(file_contents)

main()



In Python you can use the **readline** method to read a line from a file. (A line is simply a string of characters that are terminated with a \n.) The method returns the line as a string, including the \n.

Python program that uses readlines() to read from an input file - line_read.py

def main():

infile = open('grandmasters.txt', 'r')

line1 = infile.readline()

line2 = infile.readline()

line3 = infile.readline()

infile.close()

print(line1 + line2 + line3)

main()



When a file is opened for reading, a special value known as a **read position** is internally maintained for that file. A file's read position marks the location of the next item that will be read from the file.

Concatenating a Newline to a String

1. Mostly, the data items that are written to a file are not string literals, but values in memory that are referenced by variables. This would be the case in a program that prompts the user to enter data, and then writes that data to a file.

2. When a program writes data that has been entered by the user to a file, it is usually necessary to concatenate a \n escape sequence to the data before writing it. This ensures that each piece of data is written to a separate line in the file.

Python program gets string from the user, and concatenating newline to it -write_names.py

def main():

print('Enter the names of three friends.')

name1 = input('Friend #1: ')

name2 = input('Friend #2: ')

name3 = input('Friend #3: ')

myfile = open('friends.txt', 'w')

myfile.write(name1 + '\n')

myfile.write(name2 + '\n')

myfile.write(name3 + '\n')

myfile.close()

print('The names were written to friends.txt.')

main()



Reading a String and Stripping the Newline from It

The `\n` serves a necessary purpose inside a file: it separates the items that are stored in the file. However, in many cases you want to remove the `\n` from a string after it is read from a file.

The following code shows an example that uses the ***rstrip*** method to strip the `\n`

```
name = 'Joanne Manchester\n'
```

```
name = name.rstrip('\n')
```

The method returns a copy of the name string without the trailing `\n`. This string is assigned back to the name variable. The result is that the trailing `\n` is stripped away from the name string.

Appending Data to an Existing File

1. When you use the 'w' mode to open an output file and a file with the specified filename already exists on the disk, the existing file will be erased and a new empty file with the same name will be created.

2. Sometimes you want to preserve an existing file and append new data to its current contents. Appending data to a file means writing new data to the end of the data that already exists in the file.

Python program opens the file and appends data to its existing contents - append_names.py

```
def main():
    myfile = open('friends.txt', 'a')
    myfile.write('Vinka\n')
    myfile.write('Vin Diesel\n')
    myfile.write('Vino\n')
    myfile.close()
main()
```



After this program runs, the file friends.txt will contain the appended data at the end of the file.

Writing and Reading Numeric Data

Strings can be written directly to a file with the write method, but numbers must be converted to strings before they can be written. Python has a built-in function named ***str*** that converts a value to a string.

Python program to write numeric data to a file - write_numbers.py

```
def main():
    outfile = open('numbers.txt', 'w')
    num1 = int(input('Enter a number: '))
    num2 = int(input('Enter another number: '))
    num3 = int(input('Enter another number: '))
    outfile.write(str(num1) + '\n')
    outfile.write(str(num2) + '\n')
    outfile.write(str(num3) + '\n')
    outfile.close()
    print('Data written to numbers.txt')
main()
```



When you read numbers from a text file, they are always read as strings. In such a case you must convert the string to a numeric type.

Python provides the built-in function ***int*** to convert a string to an integer, and the built-in function ***float*** to convert a string to a floating-point number.

Python program to read numeric data from a file - read_numbers.py

```
def main():
    infile = open('numbers.txt', 'r')
    num1 = int(infile.readline())
    num2 = int(infile.readline())
    num3 = int(infile.readline())
    infile.close()
    total = num1 + num2 + num3
    print('The numbers are:', num1, num2, num3)
    print('Their total is:', total)
main()
```



2. Using Loops to Process Files

Files usually hold large amounts of data, and programs typically use a *loop* to process (to write or read) the data in a file.

Python program prompts the user for input and write them to the file - write_sales.py

def main():

```
    num_days = int(input('For how many days do ' + \
                          'you have sales? '))
```

```
    sales_file = open('sales.txt', 'w')
```

```
    for count in range(1, num_days + 1):
```

```
        sales = float(input('Enter the sales for day #' + \
                             str(count) + ': '))
```

```
        sales_file.write(str(sales) + '\n')
```

```
    sales_file.close()
```

```
    print('Data written to sales.txt.')
```

main()

Reading a File with a Loop and Detecting the End of the File

Quite often a program must read the contents of a file without knowing the number of items that are stored in the file. You can use a loop to read the items in the file, but you need a way of knowing when the *end of the file* has been reached.

In Python, the ***readline*** method returns an empty string (") when it has attempted to read beyond the end of a file. This makes it possible to write a *while* loop that determines when the end of a file has been reached. Here is the general algorithm, in pseudocode:

Open the file

Use readline to read the first line from the file

While the value returned from readline is not an empty string:

Process the item that was just read from the file

Use readline to read the next line from the file.

Close the file

Python program reads and displays all of the data in the file - read_sales.py

def main():

```
    sales_file = open('sales.txt', 'r')
```

```
    line = sales_file.readline()
```

```
    while line != "":
```

```
        amount = float(line)
```

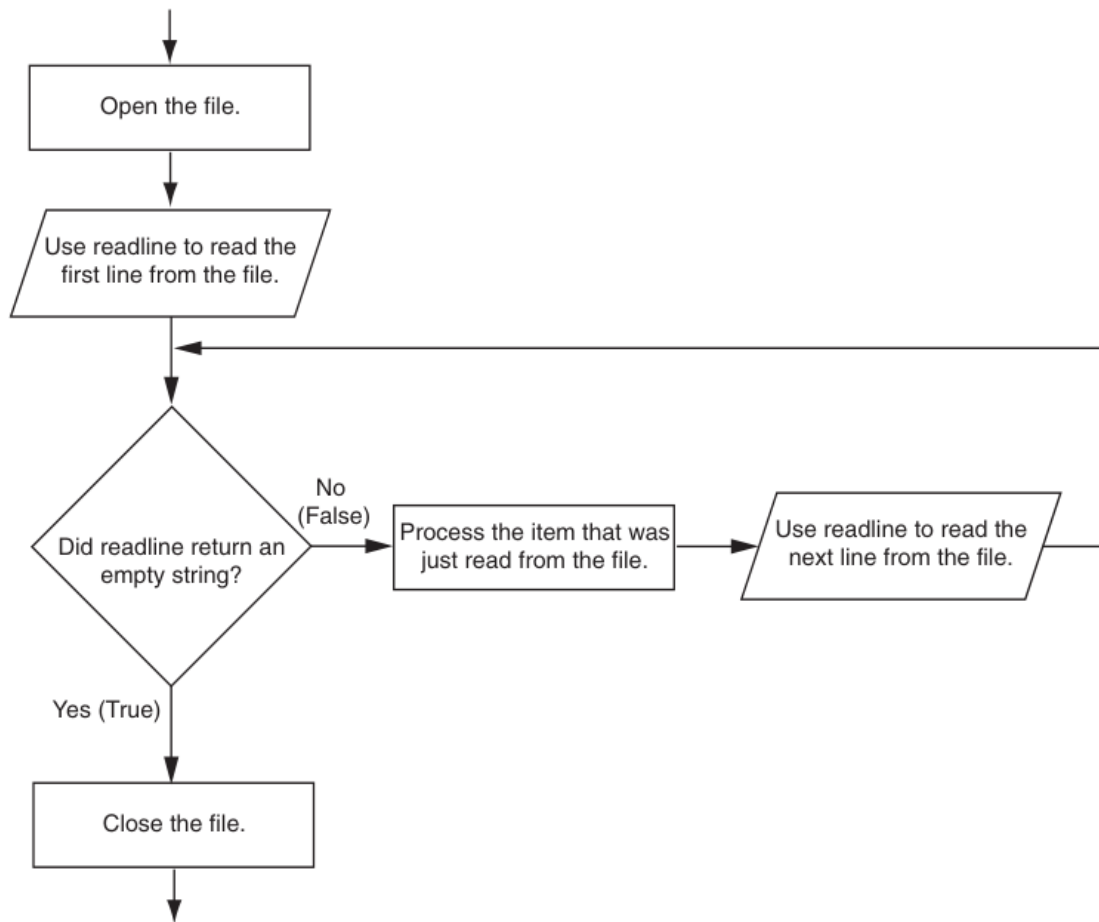
```
        print(format(amount, '.2f'))
```

```
        line = sales_file.readline()
```

```
    sales_file.close()
```

main()



General logic for detecting the end of a file using while loop**Using Python's for Loop to Read Lines**

The Python language also allows you to write a *for* loop that automatically reads line in a file without testing for any special condition that signals the end of the file. The loop does not require a *priming read* operation, and it automatically stops when the end of the file has been reached.

Here is the general format of the loop:

```

for variable in file_object:
    statement
    statement
    Etc.
  
```

In the general format, *variable* is the name of a variable and *file_object* is a variable that references a file object.

Python program reads and displays all of the data in the file - read_sales2.py

def main():

sales_file = open('sales.txt', 'r')

for line in sales_file:

amount = float(line)

print(format(amount, '.2f'))

sales_file.close()

main()

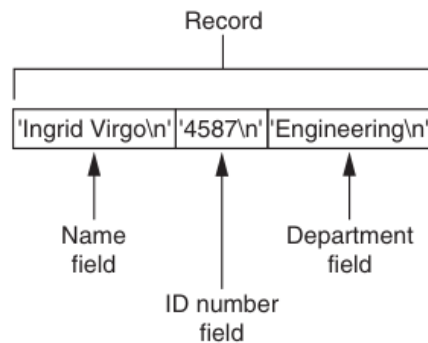


3. Processing Records

The data that is stored in a file is frequently organized in records. When data is written to a file, it is often organized into **records and fields**.

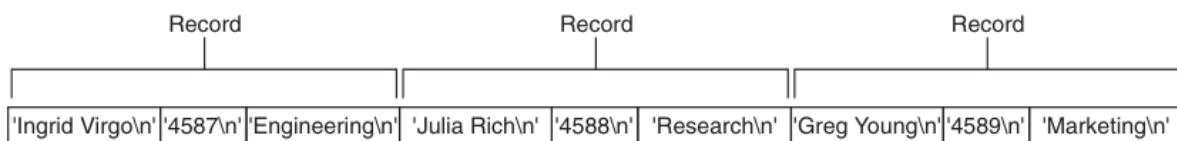
A *record* is a complete set of data about an item, and a *field* is an individual piece of data within a record.

Fields in a record



Each time you write a record to a sequential access file, you write the fields that make up the record, one after the other.

Records in a file



Python program that write records to a file - save_emp_records.py

def main():

```

    num_emps = int(input('How many employee records ' + \
        'do you want to create? '))

```

```

    emp_file = open('employees.txt', 'w')

```

```

    for count in range(1, num_emps + 1):

```

```

        print('Enter data for employee #', count, sep="")

```

```

        name = input('Name: ')

```

```

        id_num = input('ID number: ')

```

```

        dept = input('Department: ')

```

```

        emp_file.write(name + '\n')

```

```

        emp_file.write(id_num + '\n')

```

```

        emp_file.write(dept + '\n')

```

```

        print()

```

```

    emp_file.close()

```

```

    print('Employee records written to employees.txt.')

```

```

main()

```



When we read a record from a sequential access file, we read the data for each field, one after the other, until we have read the complete record.

Python program that read and display the records in the file - read_emp_records.py

```
def main():
    emp_file = open('employees.txt', 'r')
    name = emp_file.readline()
    while name != "":
        id_num = emp_file.readline()
        dept = emp_file.readline()
        name = name.rstrip("\n")
        id_num = id_num.rstrip("\n")
        dept = dept.rstrip("\n")
        print('Name:', name)
        print('ID:', id_num)
        print('Dept:', dept)
        print()
        name = emp_file.readline()
    emp_file.close()
main()
```



4. Exceptions

An *exception* is an error that occurs while a program is running, causing the program to abruptly halt. You can use the try/except statement to gracefully handle exceptions.

division.py

```
def main():
    num1 = int(input('Enter a number: '))
    num2 = int(input('Enter another number: '))
    result = num1 / num2
    print(num1, 'divided by', num2, 'is', result)
main()
```



Sample run

Enter a number: 2

Enter another number: 0

```
Traceback (most recent call last):
  File "division.py", line 7, in <module>
    main()
  File "division.py", line 4, in main
    result = num1 / num2
ZeroDivisionError: division by zero
```

The lengthy error message that is shown in the sample run is called a *traceback*. The traceback gives information regarding the line number(s) that caused the exception.

The last line of the error message shows the name of the *exception* that was raised (ZeroDivisionError) and a brief description of the error that caused the exception to be raised (integer division or modulo by zero). You can prevent many exceptions from being raised by carefully coding your program.

Python, allows you to write code that responds to exceptions when they are raised, and prevents the program from abruptly crashing. Such code is called an *exception handler*, and is written with the *try/except* statement.

General format:

```
try:
    statement      # try suite
    statement
    etc.
except ExceptionName:
    Statement      # handler
    statement
    etc.
```

When the *try/except* statement executes, the statements in the try suite begin to execute. The following describes what happens next:

1. If a statement in the *try suite* raises an exception that is specified by the *ExceptionName* in an *except* clause, then the handler that immediately follows the *except* clause executes. Then, the program resumes execution with the statement immediately following the *try/except* statement.
2. If a statement in the *try suite* raises an exception that is not specified by the *ExceptionName* in an *except* clause, then the program will halt with a *traceback* error message.
3. If the statements in the *try suite* execute without raising an exception, then any *except* clauses and *handlers* in the statement are skipped and the program resumes execution with the statement immediately following the *try/except* statement.

division2.py

```
def main():
    num1 = int(input('Enter a number: '))
    num2 = int(input('Enter another number: '))
    try:
        result = num1 / num2
        print(num1, 'divided by', num2, 'is', result)
    except ZeroDivisionError:
        print('Error: division by zero.')
```

main()

Sample run

Enter a number: 2

Enter another number: 0

Error: division by zero.

ubuntu@runcode:~/workspace\$ /bin/python /home/ubuntu/workspace/exception.py

Enter a number: 2

Enter another number: 4

2 divided by 4 is 0.5



display_file.py

Program gets the name of a file from the user and then displays the contents of the file. The program works as long as the user enters the name of an existing file. An *exception* will be raised, however, if the file specified by the user (*bad_file.txt*) does not exist.

def main():

```
filename = input('Enter a filename: ')
infile = open(filename)
contents = infile.read()
infile.close()
print(contents)
```



main()

Sample run

Enter a filename: bad_text.txt

Traceback (most recent call last):

```
File "display_file.py", line 8, in <module>
    main()
```

```
File "display_file.py", line 3, in main
    infile = open(filename, 'r')
```

FileNotFoundError: [Errno 2] No such file or directory: 'bad_text.txt'

display_file2.py

Program with a *try/except* statement that gracefully responds to an *IOError* exception.

Note: *FileNotFoundError* is a subclass of *IOError*

def main():

```
filename = input('Enter a filename: ')
try:
    infile = open(filename)
    contents = infile.read()
    infile.close()
    print(contents)
except IOError:
    print('An error occurred trying to read')
    print('the file', filename)
```



main()

Sample run

Enter a filename: good_file.py

*An error occurred trying to read
the file bad_file.txt*

Enter a filename: good_file.txt

I am a good file XD!

Handling Multiple Exceptions

In many cases, the code in a try suite will be capable of throwing more than one type of exception. In such a case, you need to write an *except* clause for each type of exception that you want to handle.

sales_report1.py

```
def main():
    total = 0.0
    try:
        infile = open('sales_data.txt', 'r')
        for line in infile:
            amount = float(line)
            total += amount
        infile.close()
        print(format(total, '.2f'))
    except IOError:
        print('An error occurred trying to read the file.')
    except ValueError:
        print('Non-numeric data found in the file.')
    except:
        print('An error occurred.')
```

***main()***

The *try suite* contains code that can raise different types of exceptions.

1. The program can raise an *IOError* exception if the *sales_data.txt* file does not exist or if it encounters a problem reading data from the file.
2. The float function can raise a *ValueError* exception if the line variable references a string that cannot be converted to a floating-point number (an alphabetic string, for example).

If an *exception* occurs in the *try* suite, the Python interpreter examines each of the *except* clauses, from top to bottom, in the *try/except* statement.

When it finds an *except* clause that specifies a type that matches the type of exception that occurred, it branches to that *except* clause.

If none of the *except* clauses specifies a type that matches the exception, the interpreter branches to the last empty *except* clause.

Using One except Clause to Catch All Exceptions

To write a *try/except* statement that simply catches any *exception* that is raised in the *try suite*, and, regardless of the exception's type, responds the same way.

You can accomplish that in a *try/except* statement by writing one *except* clause that does not specify a particular type of exception.

sales_report2.py

```
def main():
    total = 0.0
```

```

try:
    infile = open('sales_data.txt', 'r')
    for line in infile:
        amount = float(line)
        total += amount
    print(format(total, '.2f'))
    infile.close()
except:
    print('An error occurred.')

```



```
main()
```

Displaying an Exception's Default Error Message

When an exception is thrown, an object known as an *exception object* is created in memory. The exception object usually contains a *default error message* pertaining to the exception.

When you write an *except* clause, you can optionally assign the *exception object* to a variable, as shown here: ***except ExceptionName as err:***

If you want to have just one *except* clause to catch all the exceptions that are raised in a *try* suite, you can specify *Exception* as the type.

sales_report3.py

```

def main():
    total = 0.0
    try:
        infile = open('sales_data.txt', 'r')
        for line in infile:
            amount = float(line)
            total += amount
        infile.close()
        print(format(total, '.2f'))
    except Exception as err:
        print(err)

```



```
main()
```

The else Clause

The *try/except* statement may have an *optional else clause*, which appears after all the *except* clauses. Here is the *general format* of a *try/except* statement with an *else* clause:

```

try:
    #statements
except ExceptionName:
    #statements
else:
    #statements

```

The block of statements that appears after the *else* clause is known as the *else suite*. The statements in the *else suite* are executed after the statements in the *try* suite, only if no exceptions were raised. *If an exception is raised, the else suite is skipped.*

sales_report4.py

```
def main():
    total = 0.0
    try:
        infile = open('sales_data.txt', 'r')
        for line in infile:
            amount = float(line)
            total += amount
        infile.close()
    except Exception as err:
        print(err)
    else:
        print(format(total, ',.2f'))

main()
```



Note: *else clause* is executed only if the statements in the *try* suite execute without raising an *exception*.

The finally Clause

The try/except statement may have an optional finally clause, which must appear after all the except clauses. Here is the general format of a try/except statement with a finally clause:

```
try:
    #statements
except ExceptionName:
    #statements
finally:
    #statements
```

The block of statements that appears after the *finally* clause is known as the *finally suite*. The statements in the *finally* suite are always executed after the *try* suite has executed and after any *exception handlers* have executed. The statements in the *finally* suite execute whether an exception occurs or not.

The purpose of the *finally* suite is to perform cleanup operations, such as closing files or other resources. Any code that is written in the *finally* suite will always execute, *even if the try suite raises an exception.*

What If an Exception Is Not Handled?

Unless an exception is handled, it will cause the program to halt. There are two possible ways for a thrown exception to go unhandled. The first possibility is for the *try/except* statement to contain no *except* clauses specifying an exception of the right type. The second possibility is for the *exception* to be raised from outside a *try* suite. In either case, the exception will cause the program to halt.

5. Python Standard Library

Python comes with a *standard library* of functions that have already been written for you. These functions, known as library functions, Some of the functions that you have used are print, input, and range.

Some of Python's library functions are built into the Python interpreter. If you want to use one of these built-in functions in a program, you simply call the function.

Many of the functions in the standard library, however, are stored in files that are known as *modules*.

For example, one of the Python standard modules is named *math*. The math module contains various mathematical functions that work with floating-point numbers.

If you want to use any of the *math* module's functions in a program, you should write the following *import* statement at the top of the program: ***import math***

This statement causes the interpreter to load the contents of the *math module* into memory and makes all the functions in the *math* module available to the program.

Generating Random Numbers

Random numbers are useful for lots of different programming tasks. For example, *dice games*, to encrypt sensitive data.

Python provides several library functions for working with *random* numbers. These functions are stored in a module named *random* in the standard library. To use any of these functions you first need to write this import statement at the top of your program:

import random

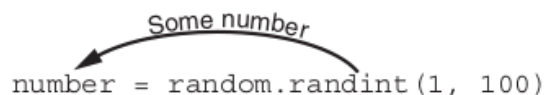
The randint Function

The following statement shows an example of how you might call the *randint* function.

```
number = random.randint(1, 100)
```

The arguments tell the function to give an integer random number in the range of 1 through 100.

The random function returns a value



```
number = random.randint(1, 100)
```

A random number in the range of
1 through 100 will be assigned to
the `number` variable.

random_numbers.py

```
import random
```

```
def main():
```

```
    number = random.randint(1, 10)
```

```
    print('The number is', number)
```

```
main()
```



The above python program generates a *random* number in the range of 1 through 10 and assigns it to the number variable.

The randrange, random, and uniform Functions

The *randrange* function takes the same arguments as the *range* function. The difference is that the *randrange* function does not return a list of values.

Instead, *it returns a randomly selected value from a sequence of values*. For example, the following statement assigns a random number in the range of 0 through 9 to the number variable:

```
number = random.randrange(10)           # end
number = random.randrange(5, 10)        # start, end
number = random.randrange(0, 101, 10)    # start, end, step
```

The *random* function, however, returns a random floating-point number. You do not pass any arguments to the *random* function. When you call it, it returns a random floating point number in the range of 0.0 up to 1.0 (but not including 1.0). Here is an example:

```
number = random.random()
```

The *uniform* function also returns a random floating-point number, but allows you to specify the range of values to select from. Here is an example:

```
number = random.uniform(1.0, 10.0)
```

In this statement the *uniform* function returns a random floating-point number in the range of 1.0 through 10.0 and assigns it to the number variable.

Random Number Seeds

1. The numbers that are generated by the functions in the *random* module are not truly random. Although we commonly refer to them as random numbers, they are actually *pseudo-random numbers* that are calculated by a formula.

2. The formula that generates random numbers has to be initialized with a value known as a *seed value*. The seed value is used in the calculation that returns the next random number in the series.

3. When the *random* module is imported, it retrieves the system time from the computer's internal clock and uses that as the seed value.

4. Because the system time changes every hundredth of a second, it is a fairly safe bet that each time you import the random module, *a different sequence of random numbers will be generated*.

5. However, there may be some applications in which you want *to always generate the same sequence of random numbers*. If that is the case, you can call the *random.seed* function to specify a seed value. Here is an example:

```
random.seed(10)
```

In this example, the value 10 is specified as the seed value. If a program calls the *random.seed* function, passing the same value as an argument each time it runs, it will *always produce the same sequence of pseudorandom numbers*.

random_numbers2.py

```
import random
random.seed(10)
# we get the same sequence of pseudorandom numbers on every run
print(random.randint(1, 100))
print(random.randint(1, 100))
print(random.randint(1, 100))
```

**The math Module**

The Python standard library's math module contains numerous functions that can be used in mathematical calculations.

These functions typically accept one or more values as arguments, perform a mathematical operation using the arguments, and return the result.

square_root.py

```
import math
def main():
    number = float(input('Enter a number: '))
    square_root = math.sqrt(number)
    print('The square root of', number, 'is', square_root)
```



main()

Sample run

Enter a number: 16

The square root of 16.0 is 4.0

Python program uses the **hypot** function to calculate the length of a right triangle's hypotenuse. (**hypotenuse.py**)

```
import math
def main():
    a = float(input('Enter the length of side A: '))
    b = float(input('Enter the length of side B: '))
    c = math.hypot(a, b)
    print('The length of the hypotenuse is', c)
```



main()

Sample run

Enter the length of side A: 4

Enter the length of side B: 3

The length of the hypotenuse is 5.0

The math.pi and math.e Values

The math module also defines two variables, pi and e, which are assigned mathematical values for pi and e. You can use these variables in equations that require their values. For example, the following statement, which calculates the area of a circle, uses pi. (Notice that we use dot notation to refer to the variable.)

```
area = math.pi * radius**2
```

6. Regular Expressions (RegEx)

A *Regular Expression* is a sequence of characters that forms a search pattern. It can be used to check if a string contains the specified search pattern.

Python has a *built-in* package called `re`, which can be used to work with Regular Expressions.

`import re`

The `re` module offers a set of functions that allows us to search a string for a match:

1. *findall* – Returns a list containing all matches
2. *search* – Returns a Match Object if there is a match anywhere in the string
3. *split* – Returns a list where the string has been split at each match
4. *sub* – Replaces one or many matches with a string

Metacharacters

Metacharacters are characters with a special meaning:

Character	Description
[]	A set of characters
\	Signals a special sequence (can also be used to escape special characters)
.	Any character (except newline character)
^	Starts with
\$	Ends with
*	Zero or more occurrences
+	One or more occurrences
?	Zero or one occurrences
{}	Exactly the specified number of occurrences
	Either or
()	Capture and group

Python program to find email addresses in the given text - regex.py*import re**text = "Hello, my email addresses are ikavin@skiff.com and immkavin.ofcl@gmail.com"**# Define a regular expression pattern for matching email addresses**pattern = r'\b[A-Za-z0-9._%+-]+\@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b'**# Search for email addresses in the text**matches = re.findall(pattern, text)**for match in matches:* *print("Found:", match)*