Aggregated Reading

**Part I Introduction to Coding Concepts Using Python**

**Chapter 1 Using Python Interactively**

Python is a very versatile language that includes a lot of built-in functionality, as well as many add-ons. Python code is ***interpreted***, which means that it is executed one line at a time. There are several environments in which Python can be used. The main ones are the Python ***interpreters***, creation of ***scripts***, and Jupyter notebooks.

• The Python interpreters allow you to enter and execute code one line at a time. • Scripts allow you to save multiple lines of code in a file and then execute them sequentially, which gives the results all at once rather than executing and getting the results one line at a time. To create a script, use an editor and save a file with an extension of .py on the filename.

• Jupyter notebooks allow combinations of text that can be formatted and executable code (and the results) in one document; the code is entered in ***cells***.

The details of installation and implementation will not be covered in this book, but it is assumed that the reader will have access to some form of Python and will be able to follow along and test the code. This book uses Python 3, which is quite different from Python 2.

Since Python is interpreted, it is possible to use Python ***interactively***. This means that you (the ***user***) enter an expression, and then Python responds with the result. What this looks like depends on the environment in which you are using Python.

In some cases, such as using the Python interpreter, you will see a ***prompt***, which is generally three greater than signs in a row, e.g.

*>>>*

When you see this prompt, you can enter an expression, e.g. 3 + 5, and then hit the Return key.

*>>> 3 + 5*

8

*>>>*

The Python interpreter responds with the result (in this case, 8), and then displays another prompt so that you can enter another expression. Note that this is the prompt for a one-line expression. When code takes multiple lines, the >>> is the first or ***primary prompt***. This is followed by lines starting with an ellipsis, which is 3 dots in a row (...), which is called the ***secondary prompt***.

In some cases, such as in Jupyter notebooks or ipython, the prompt will be the word “In” and then an integer in square brackets followed by a colon, and the response will consist of the word “Out” and then the same integer in square brackets with a colon, e.g.

In[1]: *3 + 5*

Out[1]: 8

The integers increase sequentially as you enter and execute expressions.

Some environments, such as Jupyter notebooks, allow more than one expression to be entered into a ***cell***, and then ***executed*** sequentially together.

In this text, although the actual prompt looks different in different environments, the prompt(s) will be shown as >>>. The prompt and the user input will be shown in a code font in italics, and the Python response will not have italics, e.g.

*>>> 3 + 5*

8

In order to be concise, the prompt that would appear after the result will not be shown.

Note that the prompt will be shown for short, simple codes. Longer code will instead be shown in boxes.

All expressions (for now) should begin in the first column; they should not be indented. Spacing and indentation is important in Python; following this rule will avoid some errors.

Longer expressions can be extended to another line by either putting a ‘\’ at the end of the first line, or putting the expression in parentheses. This can be extended to multiple lines, also.

*>>> 1 + 2 + 3 + \*

*...4 + 5*

15

*>>> (1 + 2 + 3*

*...+ 4 + 5)*

15

Using parentheses is generally preferred. In these examples, the addition operator (+) can be at the end of one line, or at the beginning of the next, as shown. Also, note the primary prompt >>> on the first line, and the secondary prompt … on the second line of these examples.

**1.1 Variables and Assignment Statements**

If it is desired to store a value for subsequent use, a ***variable*** can be used. Values can be stored in variables using an ***assignment statement***. The general form of an assignment statement is

*variable = expression*

where the name of the variable is on the left, the ***assignment operator*** is the single equal sign, and the expression is on the right. For simplicity we can assume that the way it works is that first the expression is evaluated, and then the value of the expression is assigned to the variable. For example, the statement

*>>> mynumber = 29 + 4*

stores the number 33 in the variable called *mynumber*. This result is not displayed by default. However, just typing the name of a variable at the prompt will display the value, e.g.

*>>> mynumber*

33

The terminology is that the assignment ***binds*** the variable name *mynumber* to the value 33. The 33 is stored in a location, and the variable *mynumber* references that location.

As we will see later, what we are calling variables here are actually ***objects***. While a variable simply stores a value, an object has both ***attributes*** such as the data stored in it, as well as functions that can operate on the data (these are called ***methods***). For now, however, we will keep it simple and refer to them as variables.

Values of some types of variables can change. Types that can change are called ***mutable***, and types that cannot change are ***immutable***. Number variables are immutable types. When the variable *mynumber* is used or displayed, it refers to the value that is stored in it. The following will take the current value of the variable *mynumber* (which is 33), subtract 2 from it, to result in the number 31, which is stored in a new location. The assignment binds the variable *mynumber* to this new location. It appears as though the value of *mynumber* has changed, and it is easiest to think of it that way. In reality, *mynumber* now refers to a new location in which 31 is stored.

*>>> mynumber = mynumber – 2*

*>>> mynumber*

31

This demonstrates that the order of operations in an assignment statement is important: first the expression on the right is evaluated, and then the result is stored in a location that is bound to the variable on the left (which may or may not be the same variable). This means that the equal sign, which is the assignment operator, does NOT mean equality! One way to read this out loud, instead of using the word “equals”, is to say “*mynumber* gets the value of *mynumber* minus 2”.

Adding a value to a variable is called ***incrementing*** the variable, and subtracting from it is called ***decrementing***.

There are rules for variable names: they must start with a letter of the alphabet or the underscore character (\_), and then can have any combination of letters, digits, and the underscore character. Spaces may not be used in variable names. It is common to use all lower case letters. Variable names should be ***mnemonic***, which means that the name should help to explain what is stored in the variable. For example, to store the radius of a circle in a variable, the variable name *radius* would be mnemonic; a variable called *x* would not be descriptive.

Python also has a default variable, which is just the underscore character. Whenever an expression is entered interactively, and the result is not stored in a variable, Python stores it in the default variable \_. This variable is reused every time just an expression is entered. For example,

*>>> 3 + 5*

8

*>>> \_ + 4*

12

*>>> \_ \* 2*

24

It is best practice, however, to avoid using this default variable.

Multiple assignments can be made in one statement, e.g.

*>>> a, b = 5, 33*

This will assign the value of 5 to the variable *a* and the value of 33 to the variable *b*. It is easier to read code in which every assignment is on a separate line, however, so this usage is generally discouraged. An exception might be assigning 0’s to several variables at once, and we will see later that there are some interesting applications for this ***simultaneous assignment***.

The word ‘variable’ means something that can change, and that is how variables are used in coding. ***Constants*** are values that cannot change during the execution of the code. Some languages provide a way to define constants, but Python does not. Instead, a value that is meant to be a constant in Python code is by convention stored in a variable named with all upper-case letters. For example, a tax rate might be defined in an assignment statement as follows.

>>> TAX\_RATE = 0.05

To delete any variable that has been created, use the **del** statement:

*>>> del mynumber*

**1.2 Built-in Functions**

Python has many built-in functions that can be used. As an example of a built-in function, we will first examine the **round** function. An example of using it is:

*>>> round(7.6)*

8

This rounds the real number 7.6 up to the integer 8.

The terminology when using a function is that the function is ***called***; the ***function call*** includes the name of the function and ***argument(s)*** that are ***passed*** to it in parentheses. If there are multiple arguments, they are separated by commas. In this example, the argument 7.6 was passed to the function named **round**. Most functions ***return*** a result; in this case, the rounded result of 8 is returned.

Another, different type of function is the **print** function. For example, the value stored in a variable can be displayed using the **print** function, as in:

*>>> val = 5 \* 2*

*>>> print(val)*

10

Of course, the value of the variable can also be displayed by just typing the name of the variable. The **print** function, however, can be used to display more than one result, e.g.

*>>> print(val, 2\*3)*

10 6

Note that a space is automatically printed between the two values.

With the **print** function, the values of the expression arguments are displayed, and it might not seem like any result is returned. The **print** function, however, actually returns the special value **None**, since it does not really need to return anything. Normally this value is not seen or used, but can be displayed by assigning the result of calling the **print** function to a variable (which is not something that would typically be done!). We will see that all Python functions return something (even if it is just None, as with **print**).

Names of functions should never be used as variable names. Although it is technically possible to do this, it would eliminate the use of the function itself (at least temporarily, until the variable is deleted). For example, if the word ‘print’ is used as a variable, it could not then be used to print anything. The second statement here would create an error message because *print* is now a variable, not a function that can be called.

*>>> print = 11*

*>>> print(8-3)*

TypeError: 'int' object is not callable

Note that the actual error message may be different in the various Python programming environments.

Deleting the *print* variable would, however, restore the use of the **print** function:

*>>> del print*

*>>> print(8-3)*

5

**1.3 Types and Operators**

There are several built-in types for expressions and variables. Each type has different operators and functions that can be used with expressions of that type.

Variables in Python are actually ***objects***, which are created from ***classes***. We will see more on objects and object oriented programming (OOP) in a later chapter.

**1.3.1 Numbers**

Simple numbers can be integers (whole numbers, with no decimal point), or real numbers (which have a decimal point). Integers are of the type **int** in Python, and real numbers are of the type **float**. There are also complex numbers, which are of the type **complex** (and are in the form a + bj or a+bJ). The type of an expression can be determined using the **type** function. For example,

*>>> print(type(1+3), type(5.2))*

<class 'int'> <class 'float'>

Python shows that the type (or ***class***) of the first expression is **int**, whereas the type of 5.2 is **float**.

Numbers can be written in scientific notation using “e”, as in 2e3, which is equivalent to 2\*103. The result is always a float.

*>>> 2e3*

2000.0

There are operators and functions that work with numbers.

We have seen some basic math operators, which include:

Addition +

Subtraction -

Multiplication \*

Negation -

The addition, subtraction, and multiplication operators are called ***binary operators***, as they operate on two ***operands***. Negation is a ***unary operator***, as it operates on only one operand.

When using these operators, if the operand(s) are integers, the result will be an integer (type **int**). If anything in the expression is a **float**, however, the result will be a **float**.

*>>> 3 + 5*

8

*>>> 5.2 – 3.2*

2.0

Other operators include:

Exponentiation \*\*

Float division /

Integer division //

Remainder division %

Exponentiation, or raising one number to a power, is accomplished using the \*\* operator. For example, 5 \*\* 2 is 52, which is 5 raised to the second power, or 5 squared, or 25.

*>>> 5 \*\* 2*

25

There are several division operators.

The / division operator always results in a **float**, regardless of the types of the operands. For example,

*>>> 5.2/2*

2.6

*>>> 6/3*

2.0

The // division operator always results in an integer (conceptually, anyway!), regardless of the types of the operands. The // operator results in the whole number part of the division, and is sometimes called ***floor division*** since it gets rid of the fractional part. The following examples show that 2 divides into 7 three times, 4 divides into 7 once, and 3 divides into 7.5 two times.

*>>> 7//2*

3

*>>> 7//4*

1

*>>> 7.5//3*

2.0

Notice the types of the results. With the expression 7.5//3, the result that is returned is conceptually the integer 2, but it is stored as a **float** and not the type **int** since the expression contains a **float**.

The % operator will show the remainder (or ***modulus***) of these divisions, e.g.

*>>> 7.5%3*

1.5

*>>> 7%4*

3

So, 3 goes into 7.5 two times, with a remainder of 1.5 (7.5-3\*2), and 4 goes into 7 one time with a remainder of 3. Again, notice the types of the results.

In addition to the built-in operators, there are math functions that are in core Python. These include rounding functions such as **round**, which we have already seen:

*>>> round(7.5)*

8

The **round** function can also round floats to a specified number of decimal places. For example, we can round 8/3 to 2 decimal places. To do this, a second argument is passed to the **round** function, which is the number of decimal places.

*>>> 8/3*

2.6666666666666665

*>>> round(8/3,2)*

2.67

**1.3.2 Boolean**

***Boolean***, or ***relational***, expressions are expressions that conceptually result in the values true or false. The type for these expressions is **bool**. There are built-in values **True** and **False** (note that these are capitalized). Expressions that result in **True** are equivalent to the integer 1, and expressions that result in **False** are equivalent to the integer 0. The binary ***relational operators***

that can be used in these expressions include:

< less than

> greater than

<= less than or equals

>= greater than or equals

== equality

!= inequality

Boolean expressions return the value of **True** or **False**.

*>>> 3 < 4*

True

*>>> 6 == 5*

False

Using the integer remainder division, we can write an expression to test whether an integer is even or odd. If an integer is even, it will be divisible by 2, so the remainder division operator would return 0. For example, 8%2 is 0, so 8%2 == 0 would be **True**. For a variable *var*, the expression var%2==0 would be **True** if the variable is even, or **False** if it is odd.

*>>> var = 16*

*>>> var % 2 == 0*

True

*>>> var = 9*

*>>> var % 2 == 0*

False

Math can be done on the result of Boolean expressions, since **True** is the same as 1 and **False** is the same as 0.

*>>> myval = 3 < 4*

*>>> myval + 7*

8

The ***logical operators*** operate on Boolean expressions; they are **not**, **and**, and **or**. The **not** operator is unary; it results in the opposite of the Boolean expression. For example, **not True** is **False**, and **not False** is **True**. The **and** and **or** operators are binary. The **or** operator returns **True**

if either or both operands are **True**. The **and** operator only returns **True** if both operands are **True**.

*>>> 3 < 7 or 2 == 4*

True

*>>> 3 < 7 and 2 == 4*

False

*>>> 3 < 7 and 2 == (3 – 1)*

True

Representing the concepts of true and false is somewhat different. For numbers, 0 represents the concept of false, but anything that is not 0 can be used to represent the concept of true. We will see examples and applications of this in later sections and chapters.

**1.3.3 Precedence Table**

Expressions in parentheses take ***precedence***, or ***priority***, over the operators. There are also precedence rules for the operators themselves, e.g., multiplication takes precedence over addition.

For the math and Boolean operators that have been covered so far, the following shows the operator precedence rules. Operators on each line have priority over the operators below them in the table.

Parentheses ( )

Exponentiation \*\*

Negation -

Multiplication and Division \*, /, //, %

Addition and Subtraction +, -

Relational <, <=, >, >=, ==, != **not not**

**and and**

**or or**

When operators have the same precedence, they are evaluated from left to right. This is called the ***associativity***.

**1.3.4 Introduction to Sequences**

Numbers and Boolean expressions are simple types that only store one value. ***Data structures*** are structured variables that store more than one value. In this section, we will briefly introduce two types of data structures, ***strings*** and ***lists***. Both of these are particular kinds of data structures called ***sequences***. With sequences, the values are put in an order and can be indexed using integer indices. Much more information on strings and lists will be covered in Chapters 5 and 7.

**1.3.4.1 Strings**

Strings in Python are text, and can be stored in either single quotes (‘Hello’) or double quotes (“Hi!”). A string is a sequence of characters. For example, the following code stores strings in variables and then prints them.

*>>> stringone = 'Hello there'*

*>>> string2 = "!!!"*

*>>> print(stringone, string2)*

Hello there !!!

Printing strings using the **print** function will not show the quotes. However, displaying the value of a string variable will. Note that although the variable *string2* is created using double quotes, the default is to display the value of a string with single quotes.

*>>> string2*

'!!!'

The type name for strings is **str**.

In some languages there is a distinction between a single character and a string of characters, but in Python they are all strings. A single character is just a string that has only one character in it.

Strings are composed of individual characters. The individual characters are numbered with integers beginning with 0. These numbers are called ***subscripts*** or ***indices***. For example, for the variable *stringone*, the 11 characters have indices from 0 through 10. In the diagram, the indices are shown above the characters in the string.

0 1 2 3 4 5 6 7 8 9 10

H e l l o t h e r e

An individual character can be obtained by putting the index in square brackets after the string variable name. This is called ***indexing*** into the string. For example, the following will print the first two characters in the string variable *stringone*.

*>>> print(stringone[0], stringone[1])*

H e

The **len** function will return the length of the string, which is the number of characters in a string, e.g.:

*>>> len(stringone)*

11

With a length of 11, the string has indices from 0 through 10.

For strings, indexing returns the individual characters in the string.

The ***empty string*** is a string that contains zero characters. An empty string can be created using quotes with nothing inside, e.g. using single quotes:

*>>> len('')*

0

The **in** operator will determine whether or not a string is in another string, and return **True** if it is or **False** if it is not. The **not in** operator will return the opposite.

*>>> 'x' in 'abcde'*

False

*>>> 'x' not in 'abcde'*

True

Strings are ***immutable***, which means that they cannot be modified. Any attempt to change character(s) in a string will result in an error message.

*>>> myword = 'Hello'*

*>>> myword[1] = 'a'*

TypeError: 'str' object does not support item assignment

There are operators for strings. To ***concatenate***, or join strings together, the ‘+’ operator is used.

*>>> 'b' + 'cde'*

'bcde'

Note that there are no spaces in between the concatenated characters.

Strings can be compared using the relational operators. String comparisons are based on the ***character encoding***. Two common encoding sequences are ASCII and Unicode. Basically, characters are put into a sequence and given equivalent integer values. In most encoding sequences the letters of the alphabet are ordered sequentially, so it is true that ‘a’ is less than ‘c’ because ‘a’ comes before ‘c’ in the encoding sequence.

*>>> 'a' < 'c'*

True

If one string is shorter than another, but they start with the same characters, the longer string is greater than the shorter string.

*>>> 'ab' < 'abc'*

True

The **print** function can print combinations of strings and numbers, e.g.:

*>>> string2 = "!!!"*

*>>> print("The number is", 33, string2)*

The number is 33 !!!

Note that blanks are automatically inserted in between the text and numbers that are printed. As we have seen, by default the contents of the strings are printed without quotes around them.

To print a string variable with quotes around it, the function **repr** can be used. This function returns the string representation of the variable (again, always in single quotes).

*>>> word = "Monty"*

*>>> print(word, repr(word))*

Monty 'Monty'

**1.3.4.1 Lists**

Lists are sequences of values. Simple lists are created by putting values in square brackets, separated by commas.

The values in lists can be different types, although it is more common for them to be the same type.

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> print(numlist, type(numlist))*

[4, 52, 33, 11, -3] <class 'list'>

*>>> mixedlist = [22, 'hello', 4 > 6]*

*>>> print(mixedlist, type(mixedlist))*

[22, 'hello', False] <class 'list'>

Notice that the type of any list is **list**. Note also that the result of the expression 4>6, **False**, is stored in the list.

Since lists are a sequence type, they can be indexed using integers.. However, unlike strings, lists are a ***mutable*** type, which means that they can be modified. The values stored in lists are called ***items*** or sometimes ***elements***. The following creates a list and then indexes into the list to get the first value (which is in element 0).

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> numlist[0]*

4

Since lists are mutable, items can be modified by indexing and assigning a new value.

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> numlist[1] = 99*

*>>> print(numlist)*

[4, 99, 33, 11, -3]

Lists can contain other sequence types, for example, strings.

*>>> strlist = ['hi', 'hello', 'ciao']*

*>>> strlist[0] = 'howdy'*

*>>> print(strlist)*

['howdy', 'hello', 'ciao']

The **len** function will return the number of elements in a list. There are 3 items in the variable *strlist*. The third item in the list (strlist[2]) is a string that has 4 elements, or characters.

*>>> len(strlist)*

3

*>>> len(strlist[2])*

4

The **in** operator will determine whether a value is in a list or not.

*>>> 4 in numlist*

True

*>>>* '*hello*' *in strlist*

True

*>>>* '*e*' *in strlist*

False

The character 'e' is not one of the elements of *strlist*, although it is in strlist[1].

**1.3.5 Type Casting**

***Type casting*** means converting an expression from one type to another. The type names in Python can be used as function names to accomplish this. For example, to convert *x* to a **float**, use **float(x)**, e.g.

*>>> float(4)*

4.0

Note that in the following, the // operator still returns the result of the integer whole number division; the integer result of 3 is then converted to the type **float**:

*>>> float(7//2)*

3.0

Converting a **float** to an **int** will truncate the decimal places.

*>>> int(5.7)*

5

A string variable that contains a number can be converted to a number type, e.g.:

*>>> strnum = '123'*

*>>> actualnum = int(strnum)*

*>>> print(actualnum, type(actualnum))*

123 <class 'int'>

Conversely, a number can be converted to a string. For example, if we wanted to know how many digits in a number, we could convert the number to a string and then use the **len** function (which does not work on numbers) to get the number of digits:

*>>> mynum = 1234*

*>>> strnum = str(mynum)*

*>>> print(len(strnum))*

4

Using the **bool** function to cast numbers to the type **bool**, we can see that numbers that are not 0 can be used to represent the concept of true.

*>>> print(bool(0), bool(1), bool(33), bool(4.2))*

False True True True

**1.4 Modules**

We have seen that Python has built-in functions. Many other functions are stored in ***modules***. Modules in Python can be used to store groups of related functions (and sometimes other definitions). Modules are separate from the core Python, and must be ***imported*** before the functions contained in them can be used. A module is imported using the **import** statement, and then functions contained in the module can be referenced using the ***dot operator***. Modules are stored in files that have a .py extension on the filename.

There are modules that are built into Python’s ***standard library***, and you can also define your own modules. Some useful standard modules include the **math** module, the **random** module, and the **statistics** module. There are also ***external library modules*** that have been developed and are widely distributed. Some of these, such as **numpy**, **pandas**, and **scipy**, contain functions

that are useful in data science applications. The **matplotlib** module has functions that help to visualize data. These modules will be introduced in Part II of this book.

The **math** module contains math functions, as well as some built-in ***constants***. Unlike variables, constants cannot change. The constants in the **math** module include:

pi 3.14159

e 2.71828

tau 6.28318

inf infinity

nan not a number

Notice that the constant **tau** is equivalent to 2\***pi**. The constant **e**, called Euler’s number, is the base for natural logarithms. Do not confuse the constant **e** here with the e used in scientific notation!

Although by convention user-defined constants are variables that are named using upper case letters, the constants that are in the **math** module are named using lower case letters. Unfortunately, they are not true constants in that their values can be modified.

To use one of them, to calculate π\*radius2, for example, we might enter the following:

*>>> import math*

*>>> radius = 3*

*>>> math.pi\*radius\*\*2*

28.274333882308138

The **import** statement imports the **math** module and then the dot operator is used as math.pi to refer to the constant **pi**.

The **round** function could be used here to reduce the displayed number of decimal places.

Note that to use the constant **pi**, it is necessary to give the name of the module, the dot operator, and then the name of the constant. In order to make this simpler, and avoid the use of the dot operator, certain or all constants and functions from a module can be imported using the **from** statement. For example, to just import **pi**:

*>>> from math import pi*

After importing **pi**, just the name **pi** can be used without first specifying **math** and the dot operator.

*>>> from math import pi*

*>>> radius = 3*

*>>> round(pi\*radius\*\*2,2)*

28.27

To import both **pi** and **e**:

*>>> from math import pi, e*

To import everything:

*>>> from math import \**

The \* is sometimes called a ***wildcard*** and in this case means everything. Importing everything from a module is not generally recommended since you probably do not know the names of everything that will be imported, and some names could interfere with names that you already have (variable names, for example).

Once the **math** module has been imported, all of the constants and functions can be seen using the **help** function.

*>>> help(math)*

The **math** module contains many useful math functions. These include trig functions such as **sin**, **cos**, and **tan**. Another function that is used a lot is the square root function, **sqrt**. There is also a function **exp**, which returns the constant **e** raised to a specified power.

To read about a particular function, use the dot operator. For example,

*>>> help(math.sqrt)*

Help on built-in function sqrt in module math:

sqrt(x, /)

Return the square root of x.

**1.5 Objects and Methods**

Python is based on objects. Objects have special functions, called ***methods***, that are associated with every type. Methods are different from functions in that, instead of explicitly passing an expression to them in parentheses, they are called implicitly with an expression using the dot operator.

For example, for strings there is a method called **upper**, which will convert all letters of the alphabet in a string to upper case.

*>>> mychars = 'Hello123'*

*>>> mychars.upper()*

'HELLO123'

This displays the letters in the string in upper case, but it does not change the variable *mychars*. (Recall that string variables are immutable; they cannot be changed.) The string variable *mychars* is not passed explicitly to the **upper** method, but rather is passed implicitly by using the dot operator. In this case, there is no need to pass any arguments so the parentheses after the method name are empty. It is necessary, however, to have the empty parentheses. Note that any non-alphabetical characters are not changed.

If it is desired to change the variable, the result of the expression must be assigned back to the variable.

*>>> mychars = 'Hello123'*

*>>> mychars = mychars.upper()*

*>>> mychars*

'HELLO123'

Similarly, there is a method called **lower** that will convert letters in a string to lower case.

In the previous example, a variable was used as the calling expression, but that is not necessary. The calling expression in this case can be just a string.

*>>> 'Hello123'.lower()*

'hello123'

Another method for strings is **index**, which returns the index of the first occurrence of a specified character, or the beginning of a specified ***substring***. Recall that the indices begin at 0.

*>>> urname = 'monty python'*

*>>> urname.index('y')*

4

*>>> urname.index('py')*

6

Note that with the **index** method, the string is passed implicitly using the dot operator, and the substring is passed explicitly in parentheses. The call to the method urname.index(‘y’) is asking for the location of the string ‘y’ in the variable *urname*.

Sometimes it is not known whether the letters in a string are lower or upper case, so they can be converted to all lower (or upper) before finding an index.

*>>> urname = 'Monty Python'*

*>>> urname.lower().index('py')*

6

Notice the use of the dot operator twice here. First, the **lower** method was used to convert the string to lower case, and then the **index** method was used to find the location of ‘py’.

There are some methods that return **True** or **False**. For example, there are “is” methods that ask a question and return either **True** or **False**. The **islower** method will test to see whether or not all letters of the alphabet in a string are lower case.

*>>> mychars = 'Hello123'*

*>>> mychars.islower()*

False

*>>> mychars = 'hello123'*

*>>> mychars.islower()*

True

Similarly, the **isupper** method will test to see whether or not all letters of the alphabet in a string are upper case.

Another method that can be used with strings is **startswith**, which determines whether or not a string starts with a particular character or string, and returns **True** if it does or **False** if it does not.

*>>> word = "Monty"*

*>>> print(word.startswith('Mo'), word.startswith('&'))* True False

In this case the **startswith** method is called with the string variable *word*, and a string is passed as an argument to test to see whether or not the variable begins with that string (of one or more characters). Similarly, there is a method **endswith**.

More string methods will be covered in Chapter 7.

There are methods for all variable types in Python, not just strings. For **float** numbers, for example, the method **is\_integer** will return **True** if the value stored is actually an integer or **False** if it is not.

*>>> num = 33.0*

*>>> print(type(num))*

<class 'float'>

*>>> num.is\_integer()*

True

There are many methods that can be used with lists. The **index** method can be used with lists as well as strings. For example,

*>>> [4, 33, 11, 5].index(11)*

2

**1.6 Random Numbers**

It is often useful to be able to generate ***random numbers***. The **random** module has several functions that facilitate this.

The **randint** function will return a random integer. Called as **randint(a,b)**, it will return a random integer in the inclusive range from a to b. For example, the following prints a random integer in the inclusive range from 2 to 10.

*>>> from random import randint*

*>>> print(randint(2,10))*

5

The **random** function returns a random real number in the range from 0 to 1, not including 1. No arguments are passed to it, so when calling it the parentheses are empty.

*>>> from random import random*

*>>> rf = random()*

*>>> rf*

0.6405322431584719

Another function in the **random** module is the **choice** function, which chooses and returns a random item from a sequence. For example, the **choice** function can choose a random character from a string.

*>>> from random import choice*

*>>> mystring = 'abc\*&def'*

*>>> choice(mystring)*

'c'

We will see examples using other sequence types in later chapters.

**1.7 Help Utility**

We have seen that the **help** function can be used to show the functions in a given module. For example, **help(math)** will display information about the **math** module once the **math** module has been imported. To find out about a particular function in a module, use the dot operator, e.g., **help(math.sqrt).**

The **help** function can also be used to display information about a particular function that is in core Python, e.g., **help(round)** will explain the usages of the **round** function.

To find out what methods can be used with expressions and variables of a particular type, pass the name of the type to the **help** function. For example, **help(str)** will display a list of the methods that can be used with strings. To find out about a particular method, use the dot operator. For example,

*>>> help(str.islower)*

Help on method\_descriptor:

islower(self, /)

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

A slightly more readable version can sometimes be obtained using a variable that has been created:

*>>> word = "Monty"*

*>>> help(word.islower)*

Help on built-in function islower:

islower() method of builtins.str instance

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

Typing just **help()**, without any arguments, brings up a help utility, which has a special prompt of help>. At this prompt, the following can be entered:

modules shows a list of all available modules

keywords shows a list of all keywords

symbols shows a list of all symbols that are used as operators

topics shows a list of all documentation topics

quit exits the help utility

***Keywords*** (also called ***reserved words***) are words that are built in to Python and can never be used as variable names. These include commands such as **import**.

**Chapter 2 Building Blocks for Programs**

We have seen that Python can be used interactively. It is also possible to gather statements into a file, which is called a ***script***. The script can then be run, or executed, all at once rather than one statement or one cell at a time. We will also use the word ***program*** to refer to code that is stored in a script. This chapter will introduce coding constructs that are frequently used in programs.

**2.1 Executing Code**

Lines of code can be executed individually, in code cells, and in scripts.

We have seen that using one of the interpreters, code can be entered at the prompt and executed immediately; the results are shown after the input.

Using an environment such as Jupyter notebooks, multiple lines of code can be entered in one cell, and then the contents of the cell can be executed. This causes each line to be executed sequentially (for now, anyway!) and all of the results are shown.

For longer programs, it is frequently useful to use an editor to type in and save a script in a file. By convention, the file will have an extension of .py.

In the remainder of this book, short expressions and simple codes will be shown using the >>> prompt. Longer code, which may be entered as a script, will be shown in boxes.

For the most part, longer codes could potentially be entered either one line at a time, in code cells, or in scripts. However, there is one difference between scripts and interactive modes. Results of expressions are shown only in interactive modes, not in scripts. In scripts, expressions must be explicitly printed in order for the results to be seen.

**2.2 Comments**

***Comments*** are completely ignored by Python when code is executed. Comments are used by the coder to explain what the code is doing. Comments are everything from the # symbol to the end of a line. For example:

*>>> age = 33 # the age of the person*

Longer comments that cover several lines can be used to describe what a program, or a part of a program, is accomplishing. For example,

# This program will calculate the area of a circle. # The program will print the area in a nice sentence format.

Longer comments can also be created using ***docstrings***, which are strings contained in triple quotes. These are used to document user-defined functions, which will be described in Chapter 6.

**2.3 Input**

Rather than building values into a program by assigning them, it is frequently useful to ask the user for values. In Python, this is accomplished using the **input** function. In order for the user (the person running the program) to know what they are supposed to be entering, the code must ***prompt*** the user. The **input** function contains the prompt. The user’s entry should then be stored in a variable, which will always be a string. For example, assume that when prompted the user enters ‘It is now 4pm’ (without the quotes):

*>>> sometext = input('Please enter something: ')*

*>>> print(sometext)*

This would appear as:

Please enter something: It is now 4pm

It is now 4pm

The user entered ‘It is now 4pm’ (again, without the quotes) and hit the Return key, which sent the entered string to be the value of the variable *sometext*.

Since the **input** function always returns a string, if a number is required the string must be converted to the appropriate number type. For example, this code

*>>> age = input('How old are you? ')*

*>>> numage = int(age)*

*>>> print('This time next year you will be', numage+1)* would result in the following if the user enters 32:

How old are you? 32

This time next year you will be 33

The age ‘32’, returned as a string, was cast to the type **int**.

It will be easiest for now to have a separate **input** statement for every value that the user needs to enter. Although it is possible to prompt for more than one value in a single **input** statement, it will all be in one string variable. Text processing would be necessary to break the string into the various parts, and to convert strings containing numbers to number types. This will be demonstrated later.

**2.4 Formatting Output**

We have seen that the **print** function can be used to display combinations of text and numbers. Some special characters can be used within the strings. The ‘\n’ character, called the ***newline character***, moves the cursor down to the next line when printed.

|  |
| --- |
| *print('See what \nthis does')*  *print('OK?')* |

See what

this does

OK?

Explicitly printing the newline character immediately moves down to the next line.

The **print** function will automatically print a newline character at the end, so in this example the second **print** function started printing on the third line.

Without passing any arguments, **print()** will simply print a newline character. In the following example, doing so prints a blank line.

|  |
| --- |
| print('\*\*\*\*')  print()  print('####') |

\*\*\*\*

####

It is also possible to end the result from a **print** statement with something other than a newline character by specifying the **end** keyword.

|  |
| --- |
| *print('See what \nthis does', end = ' ') print('OK?')* |

See what

this does OK?

In this case, the first **print** statement ended with a space rather than a newline, so the second **print** began on that same line.

To print a single quote within a string that is specified using single quotes, use \’.

*>>> print('Isn\'t this grand')*

Isn't this grand

Of course, if the string is created using double quotes, that is not necessary.

*>>> print("Isn't necessary here!")*

*Isn't necessary here!*

To print a double quote within a string that is specified using double quotes, use \”.

*>>> print("The character \"a\" is a short string")* The character "a" is a short string

In order to format the output, ***f-strings*** (short for ***formatted strings***) can be used. These are created by putting the letter f (or F) in front of the string. By putting expressions in curly braces, the values of these expressions are filled in when the result is printed. For example,

*>>> myint = 1234*

*>>> print(f'The integer is xx{myint}xx')*

The integer is xx1234xx

When printed, the curly braces are not shown but the value of the variable *myint* is displayed in that location. The xx’s are printed just to show the where the number begins and ends. Note that if we did not use the f-string, and instead just printed the text and number separately, the output would automatically have blank spaces:

*>>> print('The integer is xx', myint, 'xx')*

The integer is xx 1234 xx

So, the f-string allows us to control spacing. Inside the curly braces, after the name of the variable, we can add a colon and then ***format specifiers***. For example, ‘8d’ says to print an integer (‘d’ for integer!) in a field width of 8, meaning 8 characters altogether.

*>>> myint = 1234*

*>>> print(f'The integer is xx{myint:8d}xx')*

The integer is xx 1234xx

Notice that by default the number is right-justified within the field width of 8, so there are 4 blank spaces and then the 4-digit number. In order to left-justify the number, the less than sign is used.

*>>> myint = 1234*

*>>> print(f'The integer is xx{myint:<8d}xx')*

The integer is xx1234 xx

Similarly, strings can be printed within a specified field width using ‘s’ in the format specifier (‘s’ for string). Strings are left-justified. For example:

*>>> word = 'Hello'*

*>>> print(f'The word is xx{word:7s}xx')*

The word is xxHello xx

Strings can be right-justified using the greater than sign:

*>>> word = 'Hello'*

*>>> print(f'The word is xx{word:>7s}xx')*

The word is xx Helloxx

We will see in a later chapter that there is also a method that will right-justify a string.

For floats, there are more options for the format specifier. Using ‘f’ in the format specifier (‘f’ for float), the number of decimal places can be controlled. For example, ‘8.2f’ specifies a field width of 8 including 2 decimal places, and including the decimal point.

*>>> myfloat = 12.345*

*>>> print(f'The float is xx{myfloat:8.2f}xx')*

The float is xx 12.35xx

Notice that the .345 was rounded to 2 decimal places, .35. A total of 8 characters was printed: 3 blank spaces, then 12.35 which is 5 characters. The less than sign can be used to left-justify floats. Also, it is not necessary to specify the field width. Just using ‘.2f’ as the format specifier, 2 decimal places are printed and the field width is adjusted according to the number being printed.

*>>> myfloat = 12.345*

*>>> print(f'The float is xx{myfloat:.2f}xx')*

The float is xx12.35xx

This is usually preferable to specifying a field width, since it is more general.

For floats, two other specifiers besides ‘f’ are: ‘g’ which specifies a general format, and ‘e’ which specifies scientific notation.

*>>> myfloat = 12.345*

*>>> print(f'The float is xx{myfloat:f}xx')*

The float is xx12.345000xx

*>>> myfloat = 12.345*

*>>> print(f'The float is xx{myfloat:g}xx')*

The float is xx12.345xx

*>>> myfloat = 12.345*

*>>> print(f'The float is xx{myfloat:e}xx')*

The float is xx1.234500e+01xx

For any type, putting an equal sign in the curly braces after the name of the variable shows both the variable name and its value.

*>>> myint = 32 + 1*

*>>> print(f'{myint = }')*

myint = 33

The f-strings used here have all been in calls to the **print** function. However, f-strings are just strings that are formatted, and can be stored in variables.

*>>> dolamt = f'${123.456:.2f}'*

*>>> dolamt*

'$123.46'

**2.5 Scripts with Input and Output**

An ***algorithm*** is a sequence of steps needed in order to solve a problem. When coding, it is frequently useful to write out an algorithm first. Once that has been accomplished, the algorithm is then translated into code.

A basic algorithm for many programs is:

• Get the necessary inputs

• Calculate results

• Print or display the results

A script that accomplishes this can therefore be broken down into three basic parts, to implement each of these steps.

For example, let’s write a program that will calculate the area of a rectangle. What would the input(s) be? For a rectangle, we would need to know the length and the width. In order for the user to know that they are to enter the length and the width, they need to be prompted to do

so. In this case, the user would need to know the units of measure. This might be accomplished by printing instructions before prompting. We also need to know that the area of a rectangle is the product of the length and width. Once the result is obtained, it is good practice to print the result in a nicely formatted sentence. In order to be very informative, it would be appropriate

to print not just the area, but the values used to calculate the area.

So, our algorithm would be:

• Print instructions and include the units

• Prompt the user for the inputs:

o Prompt the user and read in the length

o Prompt the user and read in the width

• Calculate the area as length \* width

• Print the area in a formatted sentence

Now let’s implement that algorithm in a Python script.

|  |
| --- |
| # This script calculates the area of a rectangle  # Print instructions and prompt the user for the length and width  print('When prompted, please enter the length and width') print(' of a rectangle in units of meters.')  st\_length = input('Please enter the length: ')  rec\_length = float(st\_length)  st\_width = input('Please enter the width: ')  rec\_width = float(st\_width)  # Calculate the area  rec\_area = rec\_length \* rec\_width  # Print the results  print(f'For a rectangle with a length of {rec\_length:.1f} meters') print(f' and a width of {rec\_width:.1f} meters, the area') print(f' is {rec\_area:.3f} meters squared.') |

When prompted, please enter the length and width

of a rectangle in units of meters.

Please enter the length: 3.32

Please enter the width: 4.1

For a rectangle with a length of 3.3 meters

and a width of 4.1 meters, the area

is 13.612 meters squared.

**2.6 Debugging**

We all make mistakes! ***Debugging*** is finding and fixing mistakes in code. There are several basic types of mistakes:

• ***Syntax errors***: These are mistakes such as leaving off the rightmost quote or brackets, and are flagged by Python.

• ***Logic errors***: Logic errors are mistakes in reasoning. The code is correct Python code; it executes properly but the results are not correct. An example might be multiplying instead of dividing when making a calculation. Sometimes it is difficult to even realize that a logic error has been made, since no error messages will result.

• ***Execution-time errors***: These are errors that are only found when the code is executed. For example, the code might divide a number by a variable, but the variable stores a zero.

Python error messages are usually easy to understand. For example,

|  |
| --- |
| x = 4  y = float(input('Enter a number: ')) x/y |

Enter a number: 0

--------------------------------------------------------------------------- ZeroDivisionError Traceback (most recent call last) /var/folders/j9/v92c26p10979jkyb81f\_nr\_c0000gn/T/ipykernel\_83392/1454302463.p y in <module>

1 x = 4

2 y = float(input('Enter a number: '))

----> 3 x/y

ZeroDivisionError: float division by zero

The name of the error message is “ZeroDivisionError”, which is self-explanatory. The line of code in which this occurred is also highlighted with an arrow.

For cases in which it is not obvious what went wrong, a good tactic is to print the values of variables.

**Chapter 3 Selection**

It is possible in a program to choose whether statements are executed or not, or to choose between statements or sets of statements. Statements that accomplish this are called ***selection*** statements and include the **if** statement and the **if-else** statement. Python has additional selection statements including the **match** statement and **try-except**.

**3.1 If Statements**

The **if** statement chooses whether statement(s) are executed or not. The general form is

if condition:

action

# rest of code

The **if** statement starts with the reserved word **if**, then an expression that evaluates to either **True** or **False**, then a colon. The expression is frequently called a ***condition***. After that, the ***action*** of the **if** is indented. The action consists of one or more statements that are executed only if the expression evaluates to **True**. If the expression evaluates to **False**, the action is skipped.

The indentation is very important in Python. If the action consists of more than one statement, all must be indented to the same level. It is customary to indent by 4 spaces. Note: use the space bar, not the tab key!

In the following example, it is **True** that the value of the variable *num* is less than 50, so the action is executed, which prints ‘It is smaller’.

|  |
| --- |
| *print('OK')*  *num = 33*  *if num < 50:*  *print('It is smaller') print('And that is it')* |

OK

It is smaller

And that is it

If the value of *num* is changed to be something greater than or equal to 50, the action would be skipped entirely.

|  |
| --- |
| *print('OK')*  *num = 62*  *if num < 50:*  *print('It is smaller') print('And that is it')* |

OK

And that is it

In interactive environments in which cells are not used, the **if** statement will result in the use of primary and secondary prompts. For example, it might look like this:

In [1]: num = 62

In [2]: if num < 50:

...: print('It is smaller')

...:

In [3]: print('And that is it')

In this case, the primary prompt is In [n]:, and the secondary prompt is ...: Just hitting the enter key at the secondary prompt will end the action of the **if** statement.

Of course, building in the value of the variable *num* by assigning a value, and then testing it, does not really make sense! It would make much more sense to generate a random number, or to prompt the user for the value of the variable *num*, and then print (or not) based on the random number or what the user entered.

|  |
| --- |
| from random import randint num = randint(0,100)  if num < 50:  print('It is smaller') print('And that is it') |

It is smaller

And that is it

We can use the **in** operator to test to see whether a particular character is in a string, or whether a particular value is in a list.

|  |
| --- |
| if 'x' in 'abcde':  print("Yay, there is an 'x'!") print('Done.') |

Done.

Again, it would make more sense if the string was not built in. It could instead be something entered by the user.

|  |
| --- |
| urname = input('Please enter your name: ') if 'z' in urname:  print("Yay, there is a 'z'!")  print('OK.') |

Please enter your name: Frazier

Yay, there is a 'z'!

OK.

**3.2 If-else Statements**

To choose between two statements, or sets of statements, the **if-else** statement is used. The general form is

if condition:

ifaction

else:

elseaction

# rest of code

The **if-else** statement chooses between two actions, called ‘ifaction’ and ‘elseaction’ here. The way it works is that the condition is evaluated. If the value of the condition is **True**, then all of the statements in the ‘ifaction’ will be executed, and the **if-else** statement ends. If, on the other hand, the value of the condition is **False**, the statement(s) in the second action, ‘elseaction’, will be executed. The **if-else** statement chooses between executing the ‘ifaction’ or the ‘elseaction’. One of these actions, and only one, will be executed. The terminology is that ***control*** goes to the chosen action, and those statements are executed. Once the statements in the chosen action have been executed, control goes to the rest of the code after the **if-else** statement.

|  |
| --- |
| *num = input('Enter a number: ')*  *num = float(num) # convert the string to a float if num < 0:*  *print(f'{num} is a negative number')*  *else:*  *print(f'{num} is a nonnegative number')* |

In this example, the user is asked for a number. The code then prints whether it is a negative number or a nonnegative number. Here are two examples of what the output would look like:

Enter a number: -33

-33.0 is a negative number

Enter a number: 14

14.0 is a nonnegative number

In order to choose from more than one option, it is possible to ***nest* if-else** statements, which means putting one inside of another.

|  |
| --- |
| *num = input('Enter a number: ')*  *num = float(num) # convert the string to a float if num < 0:*  *print(f'{num} is a negative number')*  *else:*  *if num == 0:*  *print('It is a zero')*  *else:*  *print(f'{num} is a positive number')* |

When this is executed, the expression *num < 0* is evaluated. If that is **True**, the code prints that it is a negative number and the entire **if-else** statement ends. If, however, it is **False**, the action consisting of the second **if-else** statement is executed. The second, nested, **if-else** statement evaluates the expression *num == 0*. If that expression is **True**, it prints that it is a zero, and if not it prints that it is a positive number.

Notice the indentation. In this example, all actions were indented 4 spaces. This can be simplified using the **elif** keyword, as follows.

|  |
| --- |
| *num = input('Enter a number: ')*  *num = float(num) # convert the string to a float if num < 0:*  *print(f'{num} is a negative number')*  *elif num == 0:*  *print('It is a zero')*  *else:*  *print(f'{num} is a positive number')* |

With **elif**, it is not necessary to have **else** followed by another **if-else** statement. The indentation, which is required in Python, is also simpler using **elif**. There can be as many **elif** clauses as necessary to handle all of the possible actions. It is not a requirement to have an **else** at the very end, although it is common to do so to handle ‘none of the above’.

Note that in Python, unlike many languages, relational operators can be ***chained*** together. The expression *a < x < b*, for example, is equivalent to the expression *a < x and x < b*. The following will test whether or not a variable *num* is in the range from 5 to 10 inclusive.

|  |
| --- |
| *if 5 <= num <= 10:*  *print('In range')*  *else:*  *print('Not in range')* |

Like **if** statements, **If-else** statements can also be used with strings. For example, the following uses the **in** operator to determine whether or not the character ‘x’ is in a string variable *mystr*.

|  |
| --- |
| *if 'x' in mystr:*  *print('There is an x!') else:*  *print('Alas, no x.')* |

**If-else** statement can also be used to print the range of a random number, for example from **randint**.

|  |
| --- |
| *from random import randint*  *ri = randint(0,10)*  *print(ri, "is", end = " ")*  *if ri < 5:*  *print("in range 0 to 5 inclusive") else:*  *print("in range 5 through 10")* |

3 is in range 0 to 5 inclusive

Try running this code multiple times to see the (possibly) different results!

**Chapter 4 Loops**

Loops are statements that repeat other statements (called the ***action***). **For** loops are generally used as ***counted loops***, in which the number of repetitions is specified ahead of time. In Python, **for** loops are frequently used to perform the same operation(s) on everything stored in a sequence. **While** loops are ***conditional loops***, which repeat statements as long as an expression is **True**, or until something happens. ***Nested loops*** will also be covered; with nested loops, one loop is in the action of another.

**4.1 For loops**

**4.1.1 Loops using range**

**For** loops are used to repeat other statements (called the ***action***) a specified number of times.

The easiest way to specify how many times is to use the **range** function. The **range** function generates a sequence of numbers. Called with an integer *n*, **range(n)** creates a sequence of *n* integers. By using an ***iterator variable*** and the **in** operator along with the **range** function, we can specify how many times to repeat an action. The general form to specify repeating an action *n* times is:

for itervar in range(n):

action

# rest of code

This says to repeat the action *n* times. The iterator variable iterates through all of the numbers generated by the **range** function, and for each value of the iterator variable, the action is executed. The action can be any number of valid statements. All of the statements in the action must be indented to the same level (typically 4 spaces). For historical reasons, the iterator variable (*itervar*) is frequently named *i*.

For example, the following **for** loop specifies repeating the action of printing a single ‘!’ 5 times.

|  |
| --- |
| *for i in range(5):*  *print('!', end='') print('\nOK')* |

This produces the output:

!!!!!

OK

The iterator variable *i* iterates through the 5 values produced by the **range** function, and for each of those five values it prints a single exclamation point, on the same line. Specifying end = ‘ ’ keeps all of the exclamation points on the same line, with nothing printed in between them. After the loop, printing the newline character moves down to the next line, where ‘OK’ is printed.

Note that in this example, the iterator variable *i* was not used in the action; it simply specified how many times to execute the action. In some cases, however, using the value of the iterator variable in the action is desired.

The sequence that is produced by **range** begins with 0, and ends at n-1. For example, range(4) creates a sequence of 4 integers: 0, 1, 2, 3.

The following loop prints the integers 0 to 3 in a column, and then ‘OK’ after the loop.

|  |
| --- |
| *for i in range(4):*  *print(i)*  *print('OK')* |

0

1

2

3

OK

In this example, the iterator variable *i* iterates through the values in the sequence produced by the **range** function. For every value of the iterator variable, the action is executed. The action in this case was to print the value of *i* (and the newline character, which print does by default). So, when *i* had the value 0, 0 was printed. Then, the iterator variable *i* got the value of 1, and 1

was printed. Then, *i* got the value of 2 and 2 was printed. Then, *i* got the value of 3 and 3 was printed. Once the iterator variable has iterated through all of the values in the sequence, the loop is over. Once the action was repeated 4 times, for every value in the sequence 0, 1, 2, 3, the next statement in the code after the loop was executed, which printed ‘OK’.

The following prints the integers 0 to 2, each of which is followed by a colon, space, and ‘!’.

|  |
| --- |
| *for num in range(3): print(f'{num}: !')* |

0: !

1: !

2: !

Although frequently named *i*, the iterator variable can have any name. In this case, the iterator variable *num* iterated through the values 0, 1, and 2. The action that was executed for each value of *num* was to print *num* in a formatted line including a colon and exclamation mark.

So, again, sometimes the iterator variable is just used to specify how many times to repeat an action, as in:

|  |
| --- |
| for i in range(3): print('\*') |

\*

\*

\*

Sometimes the value of the iterator variable is used in the action, as in:

|  |
| --- |
| for i in range(3):  print(i) |

0

1

2

**4.1.2 Looping through sequences**

A **for** loop can be used in Python to iterate through the items in any sequence, and perform the same action for each one. The general form is

for itervar in sequence:

action

# rest of code

There is a variable, *itervar*, that iterates through all of the items in the sequence in order. For each item, the action is executed.

The following loops through all of the items in the list *somenums* and prints each in a sentence.

|  |
| --- |
| *somenums = [4, 33, 11]*  *for n in somenums:*  *print('The number is', n)* |

The number is 4

The number is 33

The number is 11

At the beginning of the loop, the iterator variable *n* gets the value of the first number in the list, which is 4, so the action prints ‘The number is 4’. Then, after the action has been executed in its entirety (in this case, it is just one statement), the iterator variable gets the next number in the list, 33. This is printed, and then *n* gets the value 11, which is printed. Once the action has been executed for all of the numbers in the list, the loop ends.

After the loop, the iterator variable *n* stores the last item from the list.

|  |
| --- |
| *somenums = [4, 33, 11]*  *for n in somenums:*  *print('The number is', n) print('n is ', n)* |

The number is 4

The number is 33

The number is 11

n is 11

The list does not need to be stored in a variable. The following will produce identical results.

|  |
| --- |
| *for n in [4, 33, 11]:*  *print('The number is', n) print('n is ', n)* |

The number is 4

The number is 33

The number is 11

n is 11

Another example illustrates looping through a list that stores different types of items, and displaying the type of each:

|  |
| --- |
| *mixedlist = [33, 'hi', False] for m in mixedlist:*  *print(type(m))* |

<class 'int'>

<class 'str'>

<class 'bool'>

Since strings are sequences, **for** loops can iterate through the characters in a string. For example, the following prints each character in a string followed by a space:

|  |
| --- |
| *myword = 'hello'*  *for c in myword:*  *print(c, end = ' ') print()* |

h e l l o

The last **print()** statement is used to move down to the next line (to print a newline).

Using the **choice** function, the following code uses a **for** loop to repeat the process of choosing a random item from a list 8 times, and prints each one.

|  |
| --- |
| *from random import choice numlist = [4, 52, 33, 11, -3] for n in range(8):*  *print(choice(numlist))* |

4

11

52

33

52

-3

33

11

To print 5 random integers in the range from 0 to 99 inclusive, we can first create a list and then as above choose random items from the list.

|  |
| --- |
| *from random import choice nums = list(range(100)) for n in range(5):*  *print(choice(nums))* |

10

42

60

54

12

This also works with just the **range** function; the list is not necessary.

|  |
| --- |
| *from random import choice nums = range(100)*  *for n in range(5):*  *print(choice(nums))* |

0

41

1

13

71

**4.1.3 Calculating Running Sums**

A useful application of a loop is to calculate a ***running sum***. A running sum typically starts at 0, and then numbers are added to the sum one at a time. For example, the sum 0+1+2+3+4+5 would start at 0, then 0+1 which is 1, then 1+2 which is 3, then 3 + 3 which is 6, then 6 + 4 which is 10 and finally 10 + 5 which is 15.

The following code accomplishes this, using a running sum variable *runsum*, and then printing the overall sum.

|  |
| --- |
| *runsum = 0*  *for i in range(6):*  *runsum = runsum + i print('The sum is', runsum)* |

The sum is 15

As the iterator variable *i* iterates through the values 0 through 5, each is added to the result that has already been stored in *runsum*. The action of the loop is simply to add to the running sum. Printing is only done after the loop, when the overall sum has been calculated.

In another example, the user is prompted for 4 numbers. Each of the numbers that the user enters is added to a running sum.

|  |
| --- |
| *runsum = 0*  *for i in range(4):*  *num = float(input('Enter a number: ')) runsum = runsum + num*  *print('The sum is', runsum)* |

Enter a number: 33

Enter a number: 4.5

Enter a number: -5

Enter a number: 2.8

The sum is 35.3

In this case, instead of adding the iterator variable *i* to the running sum, the numbers that the user enters into the variable *num* are added to the running sum.

A running sum can also be calculated from the numbers in a list.

|  |
| --- |
| *runsum = 0*  *for n in [4, 33, 11]:*  *runsum = runsum + n*  *print('The sum is', runsum)* |

The sum is 48

**4.2 Conditional Loops**

**4.2.1 While and while-else**

A ***loop*** is a statement that repeats other statement(s), which are called the action of the loop. A ***conditional loop*** repeats the action as long as an expression is **True**, or until something happens. In Python, the **while** statement is used as the conditional loop. The general form is:

while expression:

action

# rest of code

The **while** loop begins by evaluating the expression (which is also sometimes called the condition). If the expression is **True**, then the action is executed. So far, that is exactly like an **if** statement! But, after the action has been executed in its entirety, control goes back to the top of the loop, and the expression is evaluated again. If the expression is **True** this time, the action is executed again. Then, the expression is evaluated, and if it is **True**, the action is executed again. This continues as long as the expression is **True**. When the expression becomes **False**, the **while** loop ends and control goes to the rest of the code after the **while** loop. Just like selection statements and the **for** loop, the action can consist of any number of statements, which must be indented to the same level (typically 4 spaces).

For example, the **while** loop

|  |
| --- |
| *x = 2*  *while x < 5:*  *x = x + 1*  *print(x)*  *print('!')* |

would result in this output:

3

4

5

!

To begin with, the value of the variable *x* is 2, which is less than 5. So, the action is executed which increments *x* to get the value of 3, and this (3) is printed. 3 is less than 5, so the action is executed again, incrementing *x* to get the value of 4, and printing 4. 4 is still less than 5, so the action is executed again, incrementing *x* to get the value of 5, and printing 5. Then, when the expression is evaluated, 5 is not less than 5 so the expression is **False** and the **while** loop ends. The statement that prints '!' is after the loop, so it only executes once. Note that in this example the two statements in the action are indented 4 spaces.

Since the expression is always evaluated before the action in a **while** loop, it is possible that the action will not be executed at all. This could occur if the expression is **False** the first time that it is evaluated. The code

|  |
| --- |
| *x = 20*  *while x < 5:*  *x = x + 1*  *print(x)*  *print('!')* |

would result in this output:

!

Since 20 is not less than 5, the action of the loop is skipped and the code just prints '!'*.*

It is important that the action must change something in the expression so that eventually it becomes **False**. If this never happens, an ***infinite loop*** occurs. To exit from an infinite loop, hit Control-C in Windows or Linux, or Command-C on Macs for most Python environments. Others may be different; for example, in Jupyter notebooks it is necessary to choose Kernel and then Interrupt, or hit the square “interrupt the kernel” icon.

An optional **else** clause can be added to a **while** loop, which specifies an action to be executed when the condition becomes **False**. The general form is

while expression:

whileaction

else:

elseaction

# rest of code

For example, we could print the value of *x* that ends the action of the **while** loop:

|  |
| --- |
| *x = 2*  *while x < 5:*  *x = x + 1*  *print(x)*  *else:*  *print('We are done at ', x) print('!')* |

3

4

5

We are done at 5

!

The **else** clause only works if the condition becomes **False**. Any other method of ending the loop would not result in the execution of the **else** clause.

As we have seen, the **random** function returns a random real number in the range from 0 to 1, not including 1. As another example, we could use a **while** loop to print random real numbers as long as they are less than 0.5.

|  |
| --- |
| *from random import random rf = random()*  *while rf < 0.5:*  *print(f'{rf:.2f}')*  *rf = random()*  *print('And that is it!')* |

0.37

0.22

And that is it!

The **else** clause could be used to print the random number that ended the loop. **4.2.2 Error Checking**

When there is user input into a program, there is almost always a valid range of values. For example, if the user is prompted to enter the length of the sides of a square, the user should enter a positive number. There might be a tighter range than that; for example, it may be specified that the sides should be in the range from 5 to 7 meters. ***Error-checking*** means checking the user’s entry for errors. More specifically, error-checking generally involves continuing to prompt the user and read in the user’s entry until a valid value is entered. In Python, this can be accomplished using a **while** loop.

For example, the following code prompts the user for a positive number, and loops to continue prompting the user until the user does enter a positive number. For now, we will assume that the user enters a number, although we will see functions in Chapter 5 that will allow us to check to make sure that the user in fact entered a number.

|  |
| --- |
| *number = input('Enter a positive number: ')*  *number = float(number)*  *while (number <= 0):*  *number = input('Seriously! Enter a positive number: ') number = float(number)*  *print('Thanks for entering', number)* |

Running the code might result in the following.

Enter a positive number: -5

Seriously! Enter a positive number: -11.1

Seriously! Enter a positive number: 3

Thanks for entering 3.0

The user was prompted for a positive number, but entered -5. Since the **input** function returns a string, the user’s input is cast to the type **float**. Since that was a negative number (less than or equal to 0), the action of the loop was executed. In the action, the user was again prompted for a positive number (‘Seriously!’), but again the user entered a negative number. So, the action

was executed again but this time the user entered 3. Since 3 is not less than or equal to 0, the **False** condition causes the loop to cease executing. The code after the loop then printed the positive number that the user finally entered. Notice that prompting the user is repeated before the loop, and also in the action of the loop. This is so that there will be a new value of *number* every time the condition is evaluated.

Of course, it is possible that the user will follow instructions the first time and enter a positive number. In that case, running the code might result in the following.

Enter a positive number: 5.2

Thanks for entering 5.2

Since the condition is evaluated at the top of the loop, and it was already **False**, the action of the loop was skipped entirely.

**4.2.3 Counting in a while loop**

With conditional loops, it is not known ahead of time how many times the action of the loop will be executed. However, it is often useful to count how many times the action ended up being executed.

This can be accomplished by creating a counter variable, initializing it before the loop, and incrementing it by one in the action of the loop (so it is incremented every time the action is executed). For example, when error-checking we may want to know how many tries it took before the user entered a correct value.

|  |
| --- |
| counter = 0;  number = input('Enter a positive number: ')  number = float(number)  counter = counter + 1  while (number <= 0):  number = input('Seriously! Enter a positive number: ') number = float(number)  counter = counter + 1  print('Thanks for entering', number)  print('It took you', counter, 'tries.') |

Enter a positive number: -9

Seriously! Enter a positive number: 33 Thanks for entering 33.0

It took you 2 tries.

**4.3 Nested loops**

A ***nested loop*** is one loop inside of another. In other words, the action of a loop contains another loop. Frequently, a nested loop is a nested **for** loop, in which one **for** loop is part of the action of another **for** loop.

In the following example, a list of words is created. A **for** loop loops through all of the words, and then for each word, another **for** loop loops to print each character followed by a space.

|  |
| --- |
| *wordlist = ['hello', "hi", 'ciao'] for myword in wordlist:*  *for c in myword:*  *print(c, end = ' ')*  *print()*  *print("That's it!")* |

h e l l o

h i

c i a o

That's it!

The first loop is called the ***outer loop***. The action of the outer loop consists of two statements: • a loop to print each character and a space

• a **print()** to move the cursor down for the next word

The second loop, that loops over the characters, is called the ***inner loop***. The action of the inner loop is a single **print** statement. After the nested loop, there is another **print** statement to print “That’s it!”. The outer loop iterates over the words in the list. For each word, the inner loop iterates over the characters in that word.

So, the first time in the outer loop the variable *myword* will store ‘hello’. The inner loop variable *c* will then iterate through all of the characters in the variable *myword*, and for each it will print the character and then a space (all on the same line). After the inner loop has completed, the **print()** statement will print a newline character, and that is it for the action of the outer loop. Once this action has completed, the variable *myword* will store the second string in the list, “hi”. The inner loop will then iterate through all of its characters and print each one followed by a space. After both characters have been printed, the **print()** moves the cursor down to the next line. Finally, *myword* gets the value ‘ciao’, and the inner loop prints all of its characters on one line and then a newline at the end. Once the variable *myword* has iterated through all 3 words in the list, the outer loop is done so the code prints “That’s it!” after the nested loop has completed.

The following is an example of a nested loop, in which the outer loop repeats 3 times. Each time the action is executed, the next integer (0 then 1 then 2) is printed, followed by a colon and a space. The inner loop then prints, 5 times, a single ‘\*’. The second argument, end=*''*, keeps all of the \*’s on the same line and does not print anything between them. The **print()**

after the inner loop prints the newline character by default, so each of the 3 actions from the outer loop is on a separate line.

|  |
| --- |
| *for num in range(3):*  *print(f'{num}:', end=' ') for n in range(5):*  *print('\*', end='') print()* |

0: \*\*\*\*\*

1: \*\*\*\*\*

2: \*\*\*\*\*

The outer loop specifies that the action will be repeated 3 times. The action of the outer loop consists of 3 statements:

• A **print** statement to print the value of the outer loop variable

• A **for** loop to repeat the action of printing a single ‘\*’ 5 times on the same line • A **print** statement to move the cursor down for the next value of the outer loop variable

Instead of looping 5 times to print a ‘\*’, the following inner loop repeats *num* times, where *num* is the value of the outer loop variable (plus 1, so it repeats once, then twice, then three times).

|  |
| --- |
| *for num in range(3):*  *print(f'{num}:', end=' ') for n in range(num+1): print('\*', end='') print()* |

0: \*

1: \*\*

2: \*\*\*

If instead of printing 0, 1, and 2 in the beginning of the lines, we wanted 1, 2, and 3, the code would instead look like this:

|  |
| --- |
| *for num in range(3):*  *print(f'{num+1}:', end=' ') for n in range(num+1): print('\*', end='') print()* |

1: \*

2: \*\*

3: \*\*\*

These were examples of nested **for** loops. Nested loops can also contain **while** loops.

For example, let’s say we want the user to enter 3 positive numbers. That means we will use a **for** loop to repeat the action 3 times. Each time, we will error-check using a **while** loop to make sure that each time a positive number is entered. The algorithm is:

• Loop 3 times

o Prompt the user for a positive number

o While the number is not positive

▪ Print error message

▪ Prompt again

o Print the positive number

Here is an implementation of the algorithm:

|  |
| --- |
| for i in range(3):  number = input('Enter a positive number: ') number = float(number)  while (number <= 0):  print('Please follow directions!') number = input('Enter a positive number: ') number = float(number)  print('Thanks for entering', number)  print() |

Enter a positive number: 4

Thanks for entering 4.0

Enter a positive number: -3.3

Please follow directions!

Enter a positive number: 5.1

Thanks for entering 5.1

Enter a positive number: -6

Please follow directions!

Enter a positive number: 11

Thanks for entering 11.0

An extra **print()** was added to the action of the outer **for** loop in order to separate each of the actions of the **for** loop.

**Chapter 5 Data Structures**

This chapter will cover ***data structures***. Data structures, or ***structured variables***, are used to store more than one value. These are also called ***compound types***. We have already seen two types of data structures, **string**, which store multiple characters, and lists. Recall that strings and lists are examples of a particular type of data structure called a ***sequence***. With sequences, the values are put in an order and can be indexed using integer indices. Another simple type of sequence data structure, which is used very commonly in Python, is a ***tuple***. We will also cover more detail on the **range** function, as well as **enumerate**.

**5.1 Strings**

We have seen that strings in Python are text, and can be stored in either single quotes or double quotes. Strings are an example of a particular kind of data type called a ***sequence***. Sequences in general are collections of values (for strings, collections of characters) that can be indexed using integers. For strings, indexing returns the characters in the string.

We have seen indexing into a string starting at the beginning uses positive integer indices, beginning with 0. Negative indices can be used to index into a string from the end of a string. Using an index of -1 returns the last character in a string, regardless of how long it is.

*>>> myword = 'hello'*

*>>> myword[-1]*

'o'

For this string, with a length of 5, an index of -5 will return the first character.

*>>> myword = 'hello'*

*>>> myword[-5]*

'h'

So, note that with a string that has a length of 5, the positive indices go from 0 through 4, and the negative indices go from -1 through -5 (in reverse order).

Strings can also be ***sliced***. A slice of a string is a subset of the characters in the string. This is also sometimes called a ***substring***. The colon operator is used in a variety of formats to create string slices. The colon operator specifies the beginning index (which is included in the string that is returned), and the ending index (which is not included).

The following example returns the characters in indices 0 and 1:

*>>> myword = 'hello'*

*>>> myword[0:2]*

'he'

The following example only returns the character in index 2 (which is the second from the end):

*>>> myword = 'help'*

*>>> myword[-2:-1]*

'l'

The starting and/or ending indices can also be omitted. If the starting index is not specified, the default is 0 (the first index), and if the ending index is not specified, the default is the length of the string.

The following example is equivalent to *myword[3:5]* so it returns the characters in indices 3 and 4:

*>>> myword = 'hello'*

*>>> myword[3:]*

'lo'

The following example is equivalent to *myword[0:1]* so it returns the character in index 0:

*>>> myword = 'hello'*

*>>> myword[:1]*

'h'

This means that just using a colon, without specifying any indices, returns a slice consisting of the entire string.

*>>> myword = 'hello'*

*>>> myword[:]*

'hello'

When indexing into a string, a second colon can be used, and the third value indicates a step. For example, the following specifies a slice of a string beginning at index 3, through index 9 (since 10 is specified), in steps of 2, which means every other character, beginning with the fourth.

*>>> mystr = 'helloabcde'*

*>>> print(mystr[3:10:2])*

lace

Using a negative step means indexing into the string in reverse order, so the first index specified should be larger than the second.

*>>> mystr = 'helloabcde'*

*>>> print(mystr[7:2:-2])*

cal

Just specifying the step of 2 will return every other character, from the beginning:

*>>> mystr = 'helloabcde'*

*>>> mystr[::2]*

'hlobd'

Only specifying a step of -1 will reverse a string.

*>>> mystr = 'helloabcde'*

*>>> mystr[::-1]*

'edcbaolleh'

We have seen the concatenation operator +, which concatenates strings together. Notice that concatenating slices obtained by using the same number, first for the ending index (omitting the starting index), and then for the starting index (omitting the ending index), results in the entire string. The following concatenates ‘hel’ with ‘lo’:

*>>> myword = 'hello'*

*>>> myword[:3] + myword[3:]*

'hello'

To concatenate a string with itself n times, where n is an integer, the ‘\*’ operator is used.

*>>> news = 'cde'*

*>>> news\*3*

'cdecdecde'

Two functions, **ord** and **chr**, can be used to convert a character to its integer equivalent, and to convert an integer to its character equivalent, respectively. The function **ord** returns the Unicode code for a character, and **chr** returns a string consisting of one character:

*>>> ord('a')*

97

*>>> chr(97)*

'a'

*>>> ord('b')*

98

Although the letters of the alphabet are in sequence, there are other characters in the Unicode encoding sequence before the letters start. Also, the upper case letters are together in the encoding sequence, and the lower case letters are together, but there are other characters in between the upper and lower case letters.

*>>> ord('A')*

65

*>>> ord('B')*

66

*>>> ord('a') - ord('Z')*

7

We have seen that the **str** type has methods, including **index**.

Using **index**, and slicing, we can break a string into pieces. The following indexes into the name to find the location of the space between the first and last names, and then slices up to that location to get just the first name.

*>>> urname = 'monty python'*

*>>> wherespace = urname.index(' ')*

*>>> firstname = urname[:wherespace]*

*>>> firstname*

'monty'

Since the **input** function returns a string, text processing must be used to split the string into multiple values if multiple inputs are desired. For example, the following prompts the user for the length and width of a rectangle. It finds the location of the space in between them, and separates the string into the two values. It then converts both to numbers and calculates and prints the area of the rectangle.

*print('When prompted, enter the length and width of a rectangle')*

*lenwid = input('Separate with a space: ')*

*wherespace = lenwid.index(' ')*

*len = float(lenwid[:wherespace])*

*wid = float(lenwid[wherespace+1:])*

*print("Area = ", round(len \* wid, 2))*

When prompted, enter the length and width of a rectangle

Separate with a space: 3.1 4.4

Area = 13.64

**5.2 Lists**

Lists are sequences of values. Simple lists are created by putting values in square brackets, separated by commas. As we have seen, the values in lists can be different types, although it is more common for them to be the same type.

The **len** function returns the number of elements in a list..

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> len(numlist)*

5

An ***empty list*** can be created using square brackets with nothing inside.

*>>> el = []*

*>>> len(el)*

0

Since lists are a sequence type, they can be indexed using integers and they can be sliced.

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> numlist[0]*

4

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> numlist[-2:]*

[11, -3]

The **del** statement can delete item(s) from a list. This will actually modify the list variable.

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> del numlist[1]*

*>>> numlist*

[4, 33, 11, -3]

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> del numlist[1:3]*

*>>> numlist*

[4, 11, -3]

Assigning the empty list to item(s) is another way to delete items (besides using **del**).

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> numlist[0:2] = []*

*>>> numlist*

[33, 11, -3]

Since lists are mutable, items can be modified by either indexing or slicing and assigning new value(s).

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> numlist[1] = 99*

*>>> print(numlist)*

[4, 99, 33, 11, -3]

*>>> strlist = ['hi', 'hello', 'ciao']*

*>>> strlist[0] = 'howdy'*

*>>> print(strlist)*

['howdy', 'hello', 'ciao']

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> numlist[3:] = [7, 8, 9]*

*>>> print(numlist)*

[4, 52, 33, 7, 8, 9]

Notice that the number of items being replaced does not have to be the same as the number of items replacing them.

An entire list can be obtained using the slice operator, as in:

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> numlist[:]*

[4, 52, 33, 11, -3]

However, caution should be used with this because in some contexts, this is different from just referring to the list variable. We will see more on this in Section 6.1.

Lists of characters can be created from strings using the **list** function.

*>>> list("monty")*

['m', 'o', 'n', 't', 'y']

The **choice** function, which is in the **random** module, can be used to choose a random item from a list.

*>>> from random import choice*

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> choice(numlist)*

33

The **shuffle** function randomly organizes the items in a sequence. For example, the shuffle function is used to modify the order of a list in the following.

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> from random import shuffle*

*>>> shuffle(numlist)*

*>>> numlist*

[33, -3, 52, 11, 4]

Operators and functions can be used on lists. For example, lists can be concatenated using the + operator.

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> newnums = numlist + [25]*

*>>> print(newnums)*

[4, 52, 33, 11, -3, 25]

The concatenation operator can only concatenate one list to another (not an individual value to a list), which is why the 25 is in square brackets.

The **in** operator will determine whether or not a value is in a list, and return **True** if it is or **False** if it is not.

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> 33 in numlist*

True

Statistical functions can be called to perform operations on lists such as **min** for the minimum value in a list and **max** for the maximum value**.**

*>>> min(numlist)*

-3

*>>> max(numlist)*

52

There is also a built-in function **sum** that will sum the numbers in a list of numbers:

*>>> numbers = [5, 11, 4]*

*>>> sum(numbers)*

20

An error will occur if the list contains values, for example strings, that cannot be added using **sum**.

*>>> slist = ['hi', 'hello', 'ciao']*

*>>> sum(slist)*

TypeError: unsupported operand type(s) for +: 'int' and 'str'

Like strings, lists have an **index** method that will return the index of a specified value. For example, to find the location in the list of the largest value:

*>>> numlist = [4, 52, 33, 11, -3]*

*>>> numlist.index(max(numlist))*

1

**5.3 Methods for Sequence Types**

**5.3.1 Methods for All Sequence Types**

There are some methods that can be used for all sequence types, such as lists, tuples, and strings. These will be demonstrated here for lists.

The **count** method returns the number of occurrences of an item in a list.

*>>> mylist = list("monty python")*

*>>> print(mylist)*

['m', 'o', 'n', 't', 'y', ' ', 'p', 'y', 't', 'h', 'o', 'n']

*>>> mylist.count('o')*

2

*>>> mylist.count('x')*

0

The **index** method returns the index of the first occurrence of an item within a list, or an error message if it is not found.

*>>> mylist.index('y')*

4

*>>> mylist.index('x')*

ValueError: 'x' is not in list

**5.3.2 Methods for Mutable Sequence Types**

There are methods that can be used for all mutable sequence types. They will be demonstrated here using lists.

The following methods modify the list variable. The value that is returned is not the list, but the default value **None**. These methods will be shown in a series of statements.

The **append** method adds one item to the end of a list.

*>>> numlist = [5, 11, 33, 11, 2, -6]*

*>>> outval = numlist.append(44)*

*>>> print(numlist, outval)*

[5, 11, 33, 11, 2, -6, 44] None

The **insert** method inserts an item into the list at the specified index. The following inserts 14 into *numlist[2]*, and moves the rest of the items.

*>>> numlist.insert(2, 14)*

*>>> numlist*

[5, 11, 14, 33, 11, 2, -6, 44]

The **extend** method is used to concatenate another list to the end of the current list.

*>>> numlist.extend([17, -5])*

*>>> numlist*

*[5, 11, 14, 33, 11, 2, -6, 44, 17, -5]*

The result of this call to **extend** is the same as *numlist += [17, -5]*.

Note that if **append** had been used instead of **extend**, the result would be a nested list ( a list in which the list [17, -5] would be the last element). This is demonstrated with a list of characters.

*>>> charlist = ['x', 'q', 's']*

*>>> charlist.extend(['y', 'z'])*

*>>> charlist*

['x', 'q', 's', 'y', 'z']

*>>> charlist.append(['a', 'b'])*

*>>> charlist*

['x', 'q', 's', 'y', 'z', ['a', 'b']]

The **remove** method can be used to remove a specified item from a list. If the item is not in the list, an error message results.

*>>> numlist*

[5, 11, 14, 33, 11, 2, -6, 44, 17, -5]

*>>> numlist.remove(-6)*

*>>> numlist*

[5, 11, 14, 33, 11, 2, 44, 17, -5]

*>>> numlist.remove(100)*

ValueError: list.remove(x): x not in list

If there are multiple occurrences of the specified item, only the first will be removed from the list.

The **pop** method removes an item from a specified index in a list. Unlike the previous methods, however, it returns a value (not **None**), which is the item that has been removed. The index can be specified using an integer. If no index is given, the default is -1 (which means the last element in the list).

*>>> delval = numlist.pop(3)*

*>>> print(delval, 'is no longer in ', numlist)*

33 is no longer in [5, 11, 14, 11, 2, 44, 17, -5]

*>>> delval = numlist.pop()*

*>>> print(delval, ' is no longer in ', numlist)*

-5 is no longer in [5, 11, 14, 11, 2, 44, 17]

The **reverse** method reverses all elements in a list; modifying the list and returning **None**.

*>>> numlist.reverse()*

*>>> numlist*

[17, 44, 2, 11, 14, 11, 5]

The **clear** method removes everything from a list, so that it becomes an empty list.

*>>> numlist.clear()*

*>>> numlist*

[]

**5.3.3 List Method**

In addition to the methods from the previous section, which can be used on any mutable sequence type, the **list** type includes a method for sorting the list, **sort**. The **sort** method returns **None**, but sorts the list in place.

*>>> numlist = [5, 11, 33, 11, 2, -6]*

*>>> numlist.sort()*

*>>> numlist*

[-6, 2, 5, 11, 11, 33]

By default, it sorts in ***ascending*** order, (lowest to highest). This can be changed to ***descending*** order by setting the reverse flag to **True** in the function call:

*>>> numlist.sort(reverse=True)*

*>>> numlist*

[33, 11, 11, 5, 2, -6]

Note that there is also a built-in function in Python, **sorted**, which will return a new list that has been sorted, but does not sort the argument list in place.

*>>> origlist = list('python')*

*>>> newlist = sorted(origlist)*

*>>> print(origlist, newlist)*

['p', 'y', 't', 'h', 'o', 'n'] ['h', 'n', 'o', 'p', 't', 'y']

**5.4 Range and enumerate**

We have seen that the **range** function generates a sequence of integers and that when called as r**ange(n)**, it generates the sequence of integers from 0 through n-1. The type returned from **range** is **range**, although it can be passed to the **list** function if a list consisting of the individual items is desired.

*>>> mr = range(6)*

*>>> print(mr, type(mr))*

range(0, 6) <class 'range'>

*>>> ml = list(mr)*

*>>> print(ml, type(ml))*

[0, 1, 2, 3, 4, 5] <class 'list'>

So, again, **range** does not create a list; it creates an iterator. An interesting aspect of iterators is that the values are created one at a time; the entire iterator is not created first.

The built-in function **iter** will determine whether an argument can be iterated through or not, and if so what type of iterator the argument is. For example, the range variable *mr* is a range iterator, a list [3, 66] is a list iterator, and an integer is not an iterator.

*>>> iter(mr)*

<range\_iterator at 0x7f888bb480f0>

*>>> iter([3, 66])*

<list\_iterator at 0x7f888bb48070>

*>>> iter(33)*

TypeError: 'int' object is not iterable

By storing the iterator object in a variable, the **next** function returns the next value in the iterator, e.g.

*>>> imr = iter(mr)*

*>>> next(imr)*

0

*>>> next(imr)*

1

*>>> next(imr)*

2

When there are no more values, an error message is thrown if **next** is called again.

If two arguments are passed to the **range** function, they are the first value (instead of the default of 0) and the last (minus 1). For example, range(3,7) generates the sequence of integers 3, 4, 5, 6.

*>>> mr = range(3,7)*

*>>> ml = list(mr)*

*>>> print(ml)*

[3, 4, 5, 6]

An integer “step” value can also be specified as a third argument (instead of the default of 1).

*>>> mr = range(3,10,2)*

*>>> ml = list(mr)*

*>>> print(ml)*

[3, 5, 7, 9]

Step values can also be negative.

*>>> ml = list(range(10,2,-2))*

*>>> print(ml)*

[10, 8, 6, 4]

The **range** function is frequently used with **for** loops to specify how many times to execute the action of a loop. More generally, the form of a **for** loop is

for itervar in iterator:

action

The **enumerate** function can be used to return both an index and an item from a sequence such as a list.

*numlist = [4, 52, 33, 11, -3]*

*for i, item in enumerate(numlist):*

*print('Item', i, 'is', item)*

Item 0 is 4

Item 1 is 52

Item 2 is 33

Item 3 is 11

Item 4 is -3

Notice that this gives us two iterator variables: *i*, which is the index, and then *item*, which is the value of that item in the list.

**5.5 Tuples**

Tuples are similar to lists, but are immutable. Tuples are generally created by putting values in parentheses, separated by commas.

*>>> mytup = (2, 11, 33)*

Functions such as **len** and indexing/slicing work the same on tuples as on lists.

*>>> len(mytup)*

3

*>>> mytup[1]*

11

Parentheses are not always necessary:

*>>> newtuple = 5, 19*

*>>> newtuple*

(5, 19)

The concatenation operator can be used to join two tuples together.

*>>> mytup + newtuple*

(2, 11, 33, 5, 19)

An ***empty tuple*** is created using parentheses with nothing inside, e.g.:

*>>> emptup = ()*

*>>> len(emptup)*

0

To create a tuple with one entry, the value must be followed by a comma, and both must be in parentheses.

*>>> onetup = (7,)*

*>>> onetup*

(7,)

The parentheses and comma here are necessary. If the comma is omitted, the value is just an integer:

*>>> print(type((7,)), type((7)))*

<class 'tuple'> <class 'int'>

The type of a tuple is **tuple**, as shown.

Because of this, indexing into a tuple to get one value is not quite the same as slicing to get one value. A slice returns another tuple, even if it only has a length of 1:

*>>> mytup = (2, 11, 33)*

*>>> mytup[0:1]*

(2,)

Indexing using 0 returns just the integer 2:

*>>> mytup[0]*

2

The **list** function can be used to create a list from a tuple:

*>>> tl = list(mytup)*

*>>> tl*

[2, 11, 33]

The **tuple** function can be used to create a tuple from another sequence type such as a string or a list:

*>>> wordlist = ['howdy', 'hi']*

*>>> wordtuple = tuple(wordlist)*

*>>> chartuple = tuple(wordlist[0])*

*>>> print(wordlist, wordtuple, chartuple)*

['howdy', 'hi'] ('howdy', 'hi') ('h', 'o', 'w', 'd', 'y')

Putting values into a tuple is called ***packing***, as in:

*>>> mytup = 2, 11, 33*

or

*>>> mytup = (2, 11, 33)*

The reverse is called ***unpacking***. In order to unpack a tuple, it is necessary to know how many values are in the tuple and to have that many variables on the left-hand side of the assignment operator.

*>>> a, b, c = mytup*

*>>> print(mytup, a, b, c)*

(2, 11, 33) 2 11 33

Unpacking is also possible for other sequence types such as lists and strings.

**5.6 For Loops and Vectorized Code**

In general, in coding, **for** loops are used to iterate through the indices of data structures such as Python sequences in order to perform the same operation on every element. In Python, the general form of this is for a sequence *seq* is:

for i in range(len(seq)):

# do something with seq[i]

In Python, rather than iterating through the indices of the sequence, it is possible to iterate through the sequence itself, as in:

for item in seq:

# do something with item

Vectorizing code essentially means getting rid of loops, and instead using built-in functions and operators.

This can involve using operators such as the \* concatenation operator, functions such as **min**, **max**, **sum**, and **sorted**, and methods such as **count** and **reverse**.

Another powerful way to vectorize Python code when creating lists is to use ***list comprehensions.***

Comprehensions do not add any power to the language, but provide a convenient, succinct way to create sequences. A comprehension essentially compresses using a **for** loop to create and add expressions to a sequence into one line.

List comprehensions are the most common, but it is also possible to create other comprehensions.

The simplest general form of a list comprehension is

[expression for i in iterable]

which creates a list of the expressions for all values of i. For example,

*>>> [i \*\* 3 for i in range(5)]*

[0, 1, 8, 27, 64]

This creates a list of the cubes of all values of i in the range from 0 to 4 inclusive. Assigning this to a list variable

*>>> cubelist = [i\*\*3 for i in range(5)]*

is equivalent to

*cubelist = []*

*for i in range(5):*

*cubelist.append(i\*\*3)*

Conditionals can be added to the list comprehension to determine which expressions based on the iterator variable to include in the list. For example,

*>>> cubelisteven = [i\*\*3 for i in range(7) if i%2==0]*

creates a list of cubes of the even integers in the range from 0 to 6 inclusive, and is equivalent to:

*cubelisteven = []*

*for i in range(7):*

*if i%2 == 0:*

*cubelisteven.append(i\*\*3)*

**Chapter 6 More on Programming**

**6.1 User-Defined Functions**

We have seen many built-in functions in Python. You can also write your own functions, which are called ***user-defined functions***. Related user-defined functions can be stored in user-defined modules, which can then be imported into programs for use.

When we write a function, we write the ***function definition***. The function can then be called, much like built-in functions.

The general form of a function definition is:

def functionname(parameters):

""" Documentation string. """

# Function body

The first line is called the ***function header***. It consists of the reserved word **def**, the name of the function, ***parameters*** (if any) in parentheses, followed by a colon. The rest of the function, which is indented, is the ***function body***. This typically begins with a ***documentation string*** (***docstring***), in triple quotes. The docstring should be sentence(s) describing what the function does, beginning with a capitalized word and ending with a period. After the docstring there are the statements in the body of the function. It is common to indent the body of the function 4 spaces.

The function is called by giving the name of the function, and passing arguments in parentheses that will correspond to the parameters in the function header:

functionname(arguments)

Sometimes the arguments in the function call are referred to as ***actual parameters*** and the parameters in the function header are referred to as ***formal parameters***.

Although this is somewhat arbitrary, we will begin with two types of functions:

* Functions that calculate and return a value
* Functions that accomplish a task

Two main reasons for writing our own functions are:

* To be able to reuse code by calling the function whenever it is needed
* To write ***modular programs***

If there is code that will be repeated at different times, perhaps using different values, it can be made into a function and then called whenever needed.

Modular programs are programs that consist of a series of functions that do the actual work. The functions are called by a ***main program***, which can be implemented as either a script or function.

**6.1.1 Functions with Return**

We have seen examples of built-in functions that calculate and return a value, such as the **round** function. User-defined functions can also accomplish this using the **return** statement.

As an example, we will write a function that returns the area of a square. In order to calculate the area of a square, the function needs the length of each side of the square, so the side length must be passed as an argument to the function. Here is a function definition for a function named *sqrarea* that accomplishes this:

*def sqrarea(sidelen):*

*"""This function calculates the area of a square."""*

*sqar = sidelen\*\*2*

*return sqar*

The function can then be called by using the name of the function and passing a number for the length of a side of the square:

*>>> sqrarea(3)*

9

In the function call, the side length, 3, was passed to the parameter *sidelen* in the function. The function then squared this, and ***returned*** the result. We say that when the function is called, ***control*** is sent to the function, and the function begins executing. The **return** statement not only returns the value (in this case, 9), but also sends control out of the function.

In general a docstring describes what a function does and can be displayed using the **help** function:

*>>> help(sqrarea)*

Help on function sqrarea in module \_\_main\_\_:

sqrarea(sidelen)

This function calculates the area of a square.

The variable *sqar*, which was used in the function body, was not necessary. The **return** statement could just return an expression:

*def sqrarea(sidelen):*

*"""This function calculates the area of a square."""*

*return sidelen\*\*2*

The value returned from the function would normally be stored in a variable, e.g.

*>>> mysquare = sqrarea(3)*

*>>> mysquare*

9

In this example, one argument was passed to the function. For a function to calculate the area of a rectangle, both the length and width of the rectangle must be passed to the function, so there will be two parameters in the function header and two arguments passed in the function call:

*def rectarea(rlen, rwid):*

*"""This function calculates the area of a rectangle."""*

*return rlen \* rwid*

*>>> rectarea(2, 4)*

8

The values of the arguments are passed to the corresponding parameters in the function, so the first argument, 2, is passed to the first parameter *rlen*, and the second argument 4 is passed to the second parameter *rwid*.

Notice that the **return** statement returned the result of an expression; an intermediate variable could be used but is not necessary.

In Python, the **return** statement can only return one object. If it is desired to have a function return more than one thing, they can be stored in one data structure (such as a **tuple** or **list**), and that can be returned. In the following example, a function calculates and returns both the area and the perimeter of a square, by storing both in a list.

*def sqrarea\_perim(sidelen):*

*"""Calculates the area and perimeter of a square."""*

*sqar = sidelen\*\*2*

*sqperim = 4 \* sidelen*

*results = [sqar, sqperim]*

*return results*

*>>> sqrarea\_perim(11)*

[121, 44]

Using a tuple instead of a list is perhaps easier to understand, and it is easier to use the results.

*def sqrarea\_perim(sidelen):*

*"""Calculates the area and perimeter of a square."""*

*sqar = sidelen\*\*2*

*sqperim = 4 \* sidelen*

*return sqar, sqperim*

*>>> area, perim = sqrarea\_perim(11)*

*>>> print(area, perim)*

121 44

**6.1.2 Functions with no Return**

In many cases functions calculate and return values, but in some cases functions just accomplish a task, such as printing.

For example, the following function receives as parameters a string and an integer n, and prints the string n times in a row on one line. After that, it moves the cursor down.

*def printstrs(instr, n):*

*"""This function prints instr n times."""*

*for i in range(n):*

*print(instr,end='')*

*print()*

Here are two examples of calling the function:

*>>> printstrs('x',3)*

xxx

*>>> printstrs("Hi", 5)*

HiHiHiHiHi

Notice that the function does not have a **return** statement, since it is not calculating anything. However, all functions return something regardless of whether there is a **return** statement or not. The default value that is returned is a special value **None**, as we can see by passing a call to the function *printstrs* to the **print** function. In the following example, ‘aa’ and then the newline gets printed by the function, and then the result returned by the function call, **None**, is printed once the function stops executing and returns control.

*>>> print(printstrs('a',2))*

aa

None

The *printstrs* function prints the line ‘aa’ and returns **None**. The **print** function then prints the returned value, **None**. This is equivalent to:

*>>> x = printstrs('a',2)*

*>>> print(x)*

It is not always necessary to pass arguments to functions (whether they explicitly return something or not). For example, we may wish to have a function that just prints a set of instructions.

*def printinstruct():*

*"""This function just prints stuff."""*

*print('Please sit up')*

*print('Do not slouch!')*

*print('Chew with your mouth closed')*

*>>> printinstruct()*

Please sit up

Do not slouch!

Chew with your mouth closed

A function like this that just prints helps to facilitate modular programs. Of course, the instructions printed by a function would normally refer to actions that the user must take when running a program!

**6.1.3 Scope**

The ***scope*** of a variable or object is where that object is valid. For functions, everything that is defined in the function is valid only in that function. We say that variables defined in a function are ***local*** to that function, meaning that their scope is the function. Formally, what happens is that every function has its own ***symbol table***, which contains the names of the variables and their values. Once a function is called and control is sent to the function, its symbol table is created. The function’s symbol table only exists while the function is executing.

For example, for the *sqarea* function, the symbol table contains the local variable *sqar*, as well as the parameter *sidelen*.

*def sqrarea(sidelen):*

*"""This function calculates the area of a square."""*

*sqar = sidelen\*\*2*

*return sqar*

After the function is called, attempting to reference either *sqar* or *sidelen* will result in an error. This is because after the function stops executing, the symbol table, the variable *sqar*, and the parameter *sidelen* no longer exist.

*>>> area = sqrarea(4)*

*>>> print(sqar)*

NameError: name 'sqar' is not defined

**6.1.4 Lambda Functions**

***Anonymous functions***, called ***lambda functions*** in Python, are very short one line functions that do not require a formal function definition using **def**. Lambda functions are created using the keyword **lambda**. The general form is:

lambda argument(s): expression

The function begins with the reserved word **lambda**, then argument(s) followed by a colon and then an expression that uses the arguments; the result of the expression is returned.

If the lambda function is stored in a variable, the variable can be used to call the function. For example,

*>>> times3 = lambda num: num \* 3*

*>>> times3(11)*

33

One advantage of a lambda function is that it is shorter than a function definition and does not require the header with **def** or the **return** statement. A disadvantage is that the “body” of the function is confined to one simple expression.

**6.2 Assignment Statements and Copying**

Assignment statements basically assign a value to a variable. Another way of saying this is that the assignment binds the variable to the value.

We have seen that variables refer to locations in which values are stored. What happens when one variable is assigned to another variable, e.g.

*var2 = var1*

is that they both refer to the same location. This has interesting ramifications when modifying variables, especially for mutable types.

Assigning an expression (not another variable) to a variable in assignment statements creates a new location for the variable to reference, and stores the result of the expression in that location. Assigning another variable, however, does not create a new location; instead, both variables now refer to the same location.

Consider the following code:

*>>> x = 4*

*>>> y = x*

What happens is that the variable *x* is created, and it refers to a location in which a 4 is stored. Then, assigning *x* to *y* means that *y* is now also going to refer to the same location as *x*. We can picture this as follows:

|  |
| --- |
| y |
| x |
| 4 |

Then, assume that the following assignment statement is executed:

*>>> x = 5*

This creates a new location in which a 5 is stored, and *x* now refers to that new location. The variable *y*, however, has not been changed. It still refers to the location in which 4 is stored.

*>>> x*

5

*>>> y*

4

This may seem like a nuance, but it is very important especially for mutable types.

In the next section, operators and functions that illustrate how this works will be introduced.

**6.2.1 Variable Identities**

Before we examine assigning one variable to another, let us first revisit assignment statements and how they work. One important piece of information is that every location that a variable references is assigned a unique **identity**. The value of this identity can be found with the **id** function. In some cases the identity is the actual address of the location that the variable refers to. Regardless of whether this is true or not, every variable has a unique identity number. Do not try to make sense of the actual numbers!

*>>> varname = 11*

*>>> print(id(varname))*

140662851902064

What actually happens when we assign a value to a variable, and then reassign a different value to that variable? We can print the value of the variable and its identity to see what happens.

*>>> var1 = 5.3*

*>>> print(var1, id(var1))*

5.3 140663007004368

*>>> var1 = 11.11*

*>>> print(var1, id(var1))*

11.11 140663007003760

We can see that the value changed, but the identity did, also. This means that the second assignment did not actually modify what was stored in the original location referred to by *var1*, but rather created a new location.

This happens also if we modify the value of the variable by multiplying the current value by 2.

*>>> var1 = 5.3*

*>>> print(var1, id(var1))*

5.3 140663007003280

*>>> var1 = var1 \* 2*

*>>> print(var1, id(var1))*

10.6 140663007003888

Again, the second assignment created a new location and stored the result of the expression on the right (var1 \* 2) in that location, and *var1* now refers to the new location.

Let’s now see what happens when we assign one number variable to another. We can see the locations using the **id** function. However, when comparing two identities, it is simpler to use the **is** operator. The **is** operator can be used to determine whether two variables refer to the same location, or in other words, whether their identities are the same or not. The expression *a is b* results in **True** if a and b refer to the same location (have the same identities) or **False** if not.

*radius = 3*

*rad = radius*

*print(radius, id(radius))*

*print(rad, id(rad))*

*if rad is radius:*

*print('They are the same')*

*else:*

*print('They are not the same')*

*print()*

*radius = 5*

*print(radius, id(radius))*

*print(rad, id(rad))*

*if rad is radius:*

*print('They are the same')*

*else:*

*print('They are not the same')*

3 140437299009904

3 140437299009904

They are the same

5 140437299009968

3 140437299009904

They are not the same

We can see that after the initial assignment *rad = radius* the two variable names referred to the same location. However, the assignment *radius = 5* then created a new location, a new identifier, for the variable *radius* to refer to.

There is also an **is not** operator. The expression

a is not b

is equivalent to

not (a is b)

but is easier to read and understand.

The same behavior can be seen for other simple types like Boolean, and for immutable types such as strings and tuples, as shown in the following simplified examples:

*>>> relate = 6 < 10*

*>>> isit = relate*

*>>> print(relate, isit)*

*>>> isit = 6 > 10*

*>>> print(relate, isit)*

True True

True False

*>>> word = "hello"*

*>>> greetword = word*

*>>> print(word, greetword)*

*>>> word = "hi"*

*>>> print(word, greetword)*

hello hello

hi hello

*>>> tupone = (5, 11, 'x')*

*>>> tuptwo = tupone*

*>>> print(tupone, tuptwo)*

*>>> tupone = (4.4, 39)*

*>>> print(tupone, tuptwo)*

(5, 11, 'x') (5, 11, 'x')

(4.4, 39) (5, 11, 'x')

For lists, assigning a new list to one of the list variables results in the same behavior that has been seen with other types.

*listone = [5, 11, 'x']*

*listtwo = listone*

*print(listone, listtwo)*

*if listtwo is listone:*

*print('They are the same')*

*else:*

*print('They are not the same')*

*print()*

*listone = ['q', '?', 0]*

*print(listone, listtwo)*

*if listtwo is listone:*

*print('They are the same')*

*else:*

*print('They are not the same')*

[5, 11, 'x'] [5, 11, 'x']

They are the same

['q', '?', 0] [5, 11, 'x']

They are not the same

This is because again, all assignments create a new location for the variable to refer to. However, lists are different in that they are mutable. This means that the contents of a list can be modified. Changing the contents of a list variable is different from assigning a brand new list!

*listone = [5, 11, 'x']*

*listtwo = listone*

*print(listone, listtwo)*

*if listtwo is listone:*

*print('They are the same')*

*else:*

*print('They are not the same')*

*print()*

*listone[1] = 27*

*print(listone, listtwo)*

*if listtwo is listone:*

*print('They are the same')*

*else:*

*print('They are not the same')*

[5, 11, 'x'] [5, 11, 'x']

They are the same

[5, 27, 'x'] [5, 27, 'x']

They are the same

This means that the two list variables refer to the same location, and the contents of one of the items in the list was modified – but no new lists were created, so the variables still referred to the same location at the end of the code. This same behavior will occur any time a list is modified.

We have seen that in Python, for the types that we have covered so far, assigning one variable to another creates an alternate reference to the same location.

Beware of list slices, however! Slicing a list using just the colon is different from using the variable name.

*>>> listone = [5, 11, 'x']*

*>>> listtwo = listone*

*>>> listthree = listone[:]*

*>>> print(listone, listtwo, listthree)*

*>>> print(id(listone), id(listtwo), id(listthree))*

[5, 11, 'x'] [5, 11, 'x'] [5, 11, 'x']

140663007067328 140663007067328 140663007088192

We can see here that the variables *listone* and *listtwo* refer to the same location, but by using the slice operator and assigning the result to *listthree*, the *listthree* variable refers to a separate location. This does not happen with strings or tuples, however, because they are immutable.

*>>> strone = "hi"*

*>>> strtwo = strone*

*>>> strthree = strone[:]*

*>>> print(strone, strtwo, strthree)*

*>>> print(id(strone), id(strtwo), id(strthree))*

hi hi hi

140662877875888 140662877875888 140662877875888

**6.2.2 Shortcut Assignments**

Python has what are called ***shortcut assignment operators***. These add no power to the language, but save a little typing.

For example, the statement *num += 1* is equivalent to the statement *num = num + 1*.

*>>> num = 4*

*>>> num += 1*

*>>> num*

5

This also works for concatenation, e.g.:

*>>> str = 'hi'*

*>>> str += 'x'*

*>>> str*

'hix'

The shortcut is that instead of typing the name of the variable twice, once on the left and once on the right of the assignment operator =, we put the + to the left of the assignment operator and then only have to type the name of the variable once.

This also works for other operators, e.g. using \*= to multiply:

*>>> inumb = 3*

*>>> inumb \*= 2*

*>>> inumb*

6

This takes the current value of the variable *inumb*, multiplies by 2, and stores the result in a new location pointed to by the variable *inumb*.

Note that for mutable types, shortcut assignments modify the contents of the variable rather than creating a new variable.

*>>> list = [1, 5, 33]*

*>>> print(list, id(list))*

[1, 5, 33] 140437475761920

*>>> list += [14]*

*>>> print(list, id(list))*

[1, 5, 33, 14] 140437475761920

*>>> list = list + [29]*

*>>> print(list, id(list))*

[1, 5, 33, 14, 29] 140437475751424

**6.2.3 Simultaneous Assignments**

We have seen the simultaneous assignment statement, in which there are multiple variables on the left of the assignment operator, and an equal number of expressions on the right. For example, the following code initializes the variables *mysum* and *mycount* to have the value 0.

*>>> mysum, myproduct = 0, 1*

Simultaneous assignments can be used to exchange the values of two variables. The following code does not produce the desired result, since the assignment *a = b* will assign 10 to the variable *a*, and the original value of *a* (which was 3) is lost.

*>>> a = 3*

*>>> b = 10*

*>>> print(a,b)*

*>>> a = b*

*>>> b = a*

*>>> print(a,b)*

3 10

10 10

Generally in programming it is necessary to use a temporary variable for this.

*>>> a = 3*

*>>> b = 10*

*>>> print(a, b)*

*>>> temp = a*

*>>> a = b*

*>>> b = temp*

*>>> print(a,b)*

3 10

10 3

By storing the original value of *a* in the variable *temp*, the value is not lost and can then be assigned to *b*.

However, in Python this can be accomplished using the simultaneous assignment, since the assignments can be thought of as simultaneous and not sequential.

*>>> a, b = 3, 10*

*>>> print(a,b)*

*>>> a,b = b,a*

*>>> print(a,b)*

*3 10*

*10 3*

**6.2.4 Passing Arguments to Functions**

In Python, arguments are passed to functions using a methodology that is sometimes referred to as ***call-by-assignment***. That is, passing arguments to function parameters in Python works in the same way as assigning one variable to another variable. Passing an argument to a function parameter results in the function parameter referring to the same location as the argument.

The following example illustrates this for an argument which is an immutable type, in this case a number.

*def testcall(mynum):*

*""" Tests results of a function call with a number argument. """*

*print('Fn: ', mynum, id(mynum))*

*mynum = 14*

*print('Fn: ', mynum, id(mynum))*

*>>> num = 5*

*>>> print(num, id(num))*

*>>> print('Now function is called')*

*>>> testcall(num)*

*>>> print('Function has ended')*

*>>> print(num, id(num))*

5 140503077231024

Now function is called

Fn: 5 140503077231024

Fn: 14 140503077231312

Function has ended

5 140503077231024

In this example, the integer 5 was assigned to a variable *num*, and the value that *num* refers to and its id were printed. Then, the function was called. This causes the parameter *mynum* in the function to refer to the same location that *num* refers to. This can be seen from the result of the first **print** statement in the function. Then, a new value, 14, was assigned to *mynum* which means that *mynum* now refers to a new location in which the 14 is stored. The function ends at that point, and we can see that after the function has finished executing, the variable *num* has not changed.

Similar results will be obtained for other immutable parameter types, such as strings or tuples.

Now let’s try basically the same example, but using a list instead of a number. Since assignment statements always create a new location, the result is similar to the result when the argument is a number.

*def testcalllist(mylist):*

*""" Tests results of a fn call with a list argument. """*

*print('Fn: ', mylist, id(mylist))*

*mylist = [33, 5, 11]*

*print('Fn: ', mylist, id(mylist))*

*>>> listarg = [44, 2, -99]*

*>>> print(listarg, id(listarg))*

*>>> print('Now function is called')*

*>>> testcalllist(listarg)*

*>>> print('Function has ended')*

*>>> print(listarg, id(listarg))*

[44, 2, -99] 140503223213248

Now function is called

Fn: [44, 2, -99] 140503223213248

Fn: [33, 5, 11] 140503223315136

Function has ended

[44, 2, -99] 140503223213248

However, if the list parameter is modified within the function (instead of assigning a new list), the list argument that was passed to the function will change.

def testcalllistchg(mylist):

""" Tests results of a fn call with a list argument. """

print('Fn: ', mylist, id(mylist))

mylist.append(8)

print('Fn: ', mylist, id(mylist))

*>>> listarg = [44, 2, -99]*

*>>> print(listarg, id(listarg))*

*>>> print('Now function is called')*

*>>> testcalllistchg(listarg)*

*>>> print('Function has ended')*

*>>> print(listarg, id(listarg))*

[44, 2, -99] 140503223251712

Now function is called

Fn: [44, 2, -99] 140503223251712

Fn: [44, 2, -99, 8] 140503223251712

Function has ended

[44, 2, -99, 8] 140503223251712

In many cases, having the function modify the list variable is not a desired outcome. To protect the list from being changed, a slice consisting of the entire list can be passed instead of just the list variable:

*>>> listarg = [44, 2, -99]*

*>>> print(listarg, id(listarg))*

*>>> print('Now function is called')*

*>>> testcalllistchg(listarg[:])*

*>>> print('Function has ended')*

*>>> print(listarg, id(listarg))*

[44, 2, -99] 140503223187200

Now function is called

Fn: [44, 2, -99] 140503223252480

Fn: [44, 2, -99, 8] 140503223252480

Function has ended

[44, 2, -99] 140503223187200

**6.3 Modular Programs**

In general in programming, a modular program is one in which the program is broken down into separate tasks, and each task is implemented as a function. This facilitates the top-down design approach of taking a problem, breaking it down into pieces, and implementing each as a function. The program then consists basically of calling the functions to complete each of the tasks.

A basic algorithm for many programs is:

* Get input(s)
* Calculate result(s), using the input(s)
* Print and/or display the result(s)

So, a simple outline of many programs would be to have:

* Function(s) to get the input(s)
* Function(s) to calculate result(s) using the input(s)
* Function(s) to display the result(s)

As an example, we will write a program that will prompt the user for the cost of an item in a store, and then calculate and print the total cost with a tax of 3%. The program will call 3 functions:

* A function to prompt for and return the cost
* A function to calculate the total cost including the tax
* A function to print the total cost

Here are the function definitions.

*def getcost():*

*""" Prompts for item cost. """*

*cost = float(input('Enter the item cost: '))*

*return cost*

*def calctax(cost):*

*""" Calculates and returns cost plus tax. """*

*tax = cost \* .03*

*return cost + tax*

*def printtotcost(totcost):*

*""" Prints cost plus tax. """*

*print(f'The total cost is ${totcost:.2f}')*

The functions could be called from a script, which is frequently called a ***main program***. The program consists of calls to the functions.

*# main program*

*cost = getcost()*

*totalcost = calctax(cost)*

*printtotcost(totalcost)*

The output might look like this:

Enter the item cost: 11.11

The total cost is $11.44

Of course, to be more complete, the *getcost* function would error-check to make sure that the user enters a valid cost.

Sometimes the main program is implemented as a function rather than a script.

**6.3.1 Function Stubs**

When writing a program that consists of a script calling multiple functions, the best practice is not to write the entire program and then execute it to see the results. Frequently, there are errors. When an error is encountered in a particular function, the problem may be in that function, or it might be the result of an incorrect argument passed to the function, that was obtained from another function. A more effective method for constructing the program is to use ***function stubs***, which are place-holders for the actual functions.

The approach is:

* Sketch out the algorithm
* Decide what the functions are going to do
* Write the script, consisting of the calls to the functions (including the arguments that will get passed back and forth)
* Write function stubs for the individual functions
* Change each function stub to the actual function, one at a time

This methodical method of writing a program helps cut down on mistakes, and makes it easier to find bugs when they occur.

Function stubs should mimic what the function is eventually going to do, including using the arguments and returning the correct type(s).

For example, let’s say we are going to write a program that will prompt the user for a temperature in degrees Celsius, convert that to Fahrenheit, and print the results.

We might have functions to:

* Prompt the user for degrees C, error-checking to make sure it’s valid, and return the result
* Receive the degrees C and from that calculate and return degrees F
* Print both degrees C and degrees F in a nice sentence format

The script for the main program might look like this:

*# Convert C to F*

*degC = getdegc()*

*degF = c2f(degC)*

*printCF(degC, degF)*

Example function stubs might look like this:

*def getdegc():*

*“””Prompts user for degrees Celsius.”””*

*return 33*

*def c2f(degreesc):*

*“””Converts degrees C to degrees F”””*

*return degreesc + 5*

*def printCF(degreesc, degreesf):*

*“””Prints degrees C and degrees F”””*

*print(degreesc, degreesf)*

The idea is to execute the program with the function stubs and make sure that values are being passed back and forth correctly. Eventually, after prompting the user and error-checking, the *getdegc* function will return a positive number. So, for now, we just return a positive number. Eventually, once we figure out the conversion, the *c2f* function will convert C to F. For now, we’ll just add 5 so we know that this function is correctly receiving the degrees C and using this number to calculate and return degrees F. Then, the *printCF* function will eventually print in a nice sentence format, but for now it will just print the values.

The headers for the functions should not change. Putting the actual doc strings in the function stubs is a good idea.

Then, once that is working, modify the stubs one at a time. For example, you might start by modifying the *getdegc* function to prompt the user. Then, modify that function to include the error-checking. Going about this systematically really does cut down on errors and makes it easier to find them.

**Chapter 7 Text Processing**

We have seen data structures such as strings, lists, and tuples. We have also seen that every data type has associated with it special functions called methods.

In this chapter, we will cover more methods that can be used with strings, and also the string module. Additionally, we will introduce reading from and writing to files, using text files.

**7.1 String Processing Methods**

Text processing can be accomplished in Python using string variables, and the operators and methods associated with the type **str**. We have seen that strings can be indexed and sliced using the colon operator, and concatenated using the plus operator. Additionally, we have seen some of the string methods, including methods that manipulate strings such as **upper**, and methods that determine characteristics of strings, such as **isupper**. Since strings are a sequence type, the methods **index** and **count** can be used for strings. Additionally, there are many more methods that can be used to manipulate strings, and to determine the nature of strings. Since strings are an immutable type, methods that modify strings always return a new string; strings cannot be modified in place.

Case conversion methods

We have seen the **upper** method, which converts all letters of the alphabet in a string to upper case, and the **lower** method, which converts all letters of the alphabet in a string to lower case.

The **capitalize** method capitalizes just the first word in a string (so just the first word in a sentence, for example).

*>>> question = "how are you?"*

*>>> question.capitalize()*

'How are you?'

The value of the variable *question* has not been changed, however. To actually change the string variable, the result from the **capitalize** method would have to be assigned to *question*. This will create a new string that the variable *question* will now point to.

*>>> question = question.capitalize()*

Another useful method, **title**, will capitalize all of the words in a string.

*>>> urname = 'monty python'*

*>>> urname.title()*

'Monty Python'

The **swapcase** method converts upper to lower case, and lower to upper case.

*>>> wacky = 'hI tHERE'*

*>>> wacky.swapcase()*

'Hi There'

Spacing methods

There are methods that will delete extra spaces from strings. The **strip** method deletes both leading and trailing blanks from strings (but not from the middle of the string).

*>>> mystr = " hello there "*

*>>> len(mystr)*

18

*>>> newstr = mystr.strip()*

*>>> newstr*

'hello there'

The **lstrip** method deletes only leading blanks, and the **rstrip** method deletes only trailing blanks (the “l” and “r” in the method names stand for left and right).

*>>> newstr = mystr.lstrip()*

*>>> newstr*

'hello there '

*>>> newstr = mystr.rstrip()*

*>>> newstr*

' hello there'

There are several methods that will pad strings with extra leading and/or trailing fill characters, in order to center, left justify, or right justify the string within a specified width. These are named **center**, **ljust**, and **rjust**, respectively. By default, the fill character is a blank space. If the number of fill characters needed to center a string is odd, there will be one more trailing character than leading.

*>>> firstname = 'monty'*

*>>> firstname.center(8)*

' monty '

The fill character can be specified, e.g.

*>>> firstname.rjust(8,'$')*

'$$$monty'

The **zfill** method pads a string with leading zero (‘0’) characters in order to fill a specified width. This will, of course, make the most sense if the string contains a number.

*>>> '123.45'.zfill(10)*

'0000123.45'

Recall that an f-string can be used to fill in decimal places with 0’s.

*>>> mymoney = 123.45*

*>>> fmym = f'{mymoney:.3f}'*

*>>> fmym*

'123.450'

*>>> fmym.zfill(10)*

'000123.450'

Methods for Combining and Splitting Strings

The **join** method can be used to concatenate together all of the strings in an iterable (for example, a list). The method is called using a string which is used as the separator between the strings. In this example, the **join** method is called using a single space so each word is separated by a single space.

*>>> strlist = ['hello','hi','ciao']*

*>>> sepstr = ' '.join(strlist)*

*>>> sepstr*

'hello hi ciao'

The **split** method will take a string of words, and will split it into separate words and return a list of those words. The string to be used as the ***delimiter*** (the string between the words) can be passed to the method; otherwise, the default is whitespace. It is also possible to specify how many splits to do using the **maxsplit** argument; if this is omitted, all possible splits are made. If the delimiter is the space, all spaces will be ignored.

*>>> sent = "How are you"*

*>>> snwords = sent.split()*

*>>> snwords*

*['How', 'are', 'you']*

*>>> sent = "How are you"*

*>>> snwords = sent.split(maxsplit=1)*

*>>> snwords*

['How', 'are you']

If the delimiter is not whitespace, it is assumed that consecutive delimiters in the string are there to signify empty strings.

*>>> sent = "How\_are\_\_\_you" # one underscore, then three*

*>>> snwords = sent.split('\_')*

*>>> snwords*

['How', 'are', '', '', 'you']

The **rsplit** method works exactly like the **split** method, except that it splits from the right, not the left. This is important only if the **maxsplit** argument is specified.

*>>> sent = "How are you doing"*

*>>> snrspl = sent.rsplit(maxsplit=2)*

*>>> snrspl*

['How are', 'you', 'doing']

The **splitlines** method will split lines in a string, using the newline character as the delimiter, and return a list consisting of the individual lines as strings. Note that the \n is not included in the resulting strings.

*>>> longstr = "Welcome!\nHow are you?\nI am fine, thank you."*

*>>> linelist = longstr.splitlines()*

*>>> linelist*

['Welcome!', 'How are you?', 'I am fine, thank you.']

The **partition** method splits a string at the first occurrence of a specified separator and returns 3 values as a tuple: the part of the string to the left of the separator, the separator, and the part of the string to the right of the separator. The separator must be passed; there is no default for it.

*>>> sent = "How are you"*

*>>> left, sep, right = sent.partition(' ')*

*>>> print(repr(left), repr(sep), repr(right))*

'How' ' ' 'are you'

The **rpartition** method also splits a string into a tuple of 3 values, but it splits the string at the last occurrence of the specified separator, rather than the first.

*>>> sent = "How are you"*

*>>> left, sep, right = sent.rpartition(' ')*

*>>> print(repr(left), repr(sep), repr(right))*

'How are ' ' ' 'you'

Methods for Finding and Replacing

We have seen the **index** method, which returns the index of the first occurrence of the beginning of a substring within a string. If the substring is not found, the **index** method throws an error message. To protect from this, the **in** operator can be used in an **if-else** statement to check to see whether or not the substring is in the string.

*urname = 'monty python'*

*if 'x' in urname:*

*where = urname.index('x')*

*else:*

*print('x is not in the string')*

The **find** method also returns the index of the first occurrence of the beginning of a substring within a string, but if the substring is not found, it returns -1.

*>>> urname = 'monty python'*

*>>> urname.find('x')*

-1

There are similar methods, **rindex** and **rfind**, that return the index of the last occurrence of the beginning of a substring within a string. If the substring is not found, **rindex** throws an error message, whereas **rfind** returns -1.

*>>> urname = 'monty python'*

*>>> urname.rfind('o')*

10

*>>> urname = 'monty python'*

*>>> urname.rfind('x')*

-1

The **replace** method finds occurrences of a substring within a string, and replaces them with another substring. By default, all occurrences are replaced, but it is also possible to pass to the **replace** method a count of the number of occurrences to replace.

*>>> origstr = 'xxHelloxxxtherex'*

*>>> newstr = origstr.replace('x',' ')*

*>>> newstr*

' Hello there '

*>>> origstr = 'xxHelloxxxtherex'*

*>>> newstr = origstr.replace('x',' ',3)*

*>>> newstr*

' Hello xxtherex'

Boolean Methods

There are methods that ask questions about strings, and return **True**/**False** answers. We have seen the start **startswith** method, which determines whether or not a string begins with a substring, and the **endswith** method, which determines whether or not a string ends with a substring. We have also seen the **isupper** method, which determines whether or not all letters of the alphabet in a string are upper case, and the **islower** method, which determines whether or not all letters of the alphabet in a string are lower case. There are quite a few other “is” methods that ask questions and return True/False answers.

The **isalpha** method returns **True** if all of the characters in a non-empty string are alphabetic, or **False** if not. If the string is empty, it returns **False**.

*>>> testalnum = 'abcde123'*

*>>> testalnum.isalpha()*

False

*>>> testal = 'xyz'*

*>>> testal.isalpha()*

True

*>>> ''.isalpha()*

False

The **isalnum** method returns **True** if all characters in a non-empty string are alphanumeric (alphabetic or numeric), or **False** if not.

*>>> testalnum.isalnum()*

True

*>>> 'abc12!?'.isalnum()*

False

The **isdigit** method returns **True** if all of the characters in a non-empty string are digits, or **False** if not.

*>>> '45.67'.isdigit()*

False

*>>> '45678'.isdigit()*

True

String Module

There is a **string** module that contains several useful string constants. The constant **ascii\_lowercase** is a string with all of the lower case letters of the alphabet, in sequence.

*>>> import string*

*>>> string.ascii\_lowercase*

'abcdefghijklmnopqrstuvwxyz'

Similarly, the constant **ascii\_uppercase** is a string containing all of the upper case letters of the alphabet, and **ascii\_letters** is a string constant that contains all of the letters, first lower and then upper case.

*>>> string.ascii\_letters*

'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

The constant **digits** contains all of the digits.

*>>> from string import digits*

*>>> digits*

'0123456789'

The following example checks each character in a string to determine whether it is a letter of the alphabet, a digit, or something else.

*import string*

*mychars = "Hi, 33!"*

*for onechar in mychars:*

*if onechar in string.ascii\_letters:*

*print(onechar, 'is a letter!')*

*elif onechar in string.digits:*

*print(onechar, 'is a digit!')*

*else:*

*print(onechar, 'is something else.')*

H is a letter!

i is a letter!

, is something else.

is something else.

3 is a digit!

3 is a digit!

! is something else.

**7.2 Text Files**

So far, our codes have had input from the user, and the output has been displayed. It is also possible to read data from a file, and it is possible to write results to a file.

When working with files, there are three basic steps:

1. Open the file
2. Use the file
3. Close the file

A file can be opened to read from it, write to it (which means from the beginning of the file), or ***append*** to it (which means writing to it from the end; adding to an existing file). Writing, reading, and appending are called ***modes***. The function that opens a file is called **open**. The general form of calling it is:

*>>> f = open(“filename”, mode)*

where the file to be opened is in the correct folder so that the open function can locate it. The mode is indicated by a string:

* ‘r’ for reading (which is the default)
* ‘w’ for writing
* ‘a’ for appending
* ‘r+’ to both read from and write to a file

Files that have been opened and used must be closed using the **close** method:

*>>> f.close()*

**7.2.1 Writing to a file**

To write to a file, there are two options: writing to a file, and appending to a file. If the file is opened for writing using the mode ‘w’, lines are written to the file from the beginning. If the file already existed, the contents of the file are erased. If the mode ‘a’ is specified, that assumes that the file already exists, and the writing will begin at the end of the file. There are two main methods that can be used:

* **write(s)** to write a string *s* to the file
* **writelines(ls)** to write a list *ls* to the file

The **writelines** method writes an entire list of strings to a file, one string per line in the file.

The more practical method to use is **write**, which writes one string to a file. Since the **write** method only writes one string to a file, it is generally in a loop (either a **while** loop or a **for** loop). Note that the newline character will usually be at the end of each string, so that every string is written to a separate line in the file. For example, the following opens a file for writing, creates a file with 3 lines in it, and then closes the file.

f = open("tryit.txt", 'w')

for i in range(3):

f.write("Hello\n")

f.close()

**7.2.2 Reading from a file**

To read from a file once it has been opened, there are three main methods that can be used:

* **read** to read in the entire file as one string
* **readlines** to read all of the lines from the file (into a list of strings)
* **readline** to read one line from the file as a string

Since files can be very large, it is generally impractical to use **read** or **readlines**. The **readline** method is normally used. However, with the short file we created in the previous section, we will demonstrate all of these methods. The **read** method reads in all characters, including the newline characters into a string. Below we print just the first 8 characters in the string, which includes a ‘\n’. (Note: the newline character ‘\n’ is a single character).

f = open("tryit.txt")

onestr = f.read()

print(onestr[0:8])

f.close()

Hello

He

The **read** method returns an empty string if the end of file has been reached.

f = open("tryit.txt")

onestr = f.read()

print(onestr)

onestr = f.read()

print(repr(onestr))

Hello

Hello

Hello

''

The **readlines** method reads all lines from the file into a list of strings.

f = open("tryit.txt")

allstrs = f.readlines()

allstrs

['Hello\n', 'Hello\n', 'Hello\n']

The **readlines** method returns an empty list if the end of file has been reached.

The **readline** method reads one line at a time into a string. Since we know that our file has 3 lines in it, we will loop to use readline 4 times.

f = open("tryit.txt")

for i in range(4):

oneline = f.readline()

print(repr(oneline))

'Hello\n'

'Hello\n'

'Hello\n'

''

As can be seen from the output, the **readline** method returns an empty string if the end of file has been reached.

In general, when reading from a file, the procedure is to read one line at a time and process it until the end of the file has been reached. The general form that accomplishes this using a **while** loop is:

*f = open(“filename”)*

*aline = f.readline()*

*while aline:*

*# do something with the line*

*aline = f.readline()*

*f.close()*

Note that the mode ‘r’ does not need to be specified since reading is the default. Also, since an empty string is a way of representing **False**, this loops until the **readline** method returns an empty string, meaning that the end of the file has been reached.

In Python, this can also be accomplished using a **for** loop that iterates through the lines in a file object.

*f = open(“filename”)*

*for aline in f:*

*# do something with the line*

*f.close()*

Whether reading or writing, files that have been opened should always be closed after the file processing has been completed. It is easy to forget to do that, however! An alternative is to use the **with** statement, which automatically closes the file. The general form is:

*with open(filename) as f:*

*# code to process file*

The following example uses **with** for both writing to a file and reading from a file.

*with open("greets.txt", 'w') as f:*

*f.write("Hello\n")*

*f.write("Hi\n")*

*f.write("Ciao\n")*

*with open("greets.txt") as f:*

*for aline in f:*

*print(aline.rstrip())*

Hello

Hi

Ciao

The **rstrip** method was used to strip the ‘\n’ from the end of each string before printing it; otherwise, there would be blank lines between each in the output.

**Part II Data Science Libraries**

**Chapter 10 Introduction to NumPy and Arrays**

The **NumPy** module is used extensively by Python programmers for mathematical and scientific computing. Most of the **NumPy** library is based on ***arrays***, which are data structures that store values (***elements***) that are all the same type. Typically the elements in arrays are numbers, either integers or floats. The **numpy** module has many built-in functions and the array objects have many associated methods for working with numbers to facilitate applications such as linear algebra and statistics. Many data science applications work extensively with the **numpy** module.

Since it is a module, **numpy** must first be imported. It is very common to rename it as *np*:

*>>> import numpy as np*

The rest of this chapter will assume that the **numpy** module has been imported in this manner.

**10.1 Array Basics**

Arrays have ***dimensions***. An array with one dimension is either a row or a column. Sometimes these are called vectors, either a ***row vector*** or a ***column vector***. Like other sequences in Python such as lists, arrays can be indexed, and the indices begin at 0.

A ***one dimensional array*** which is a row vector containing 4 elements might be depicted as:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
| 33 | 11 | -5 | 14 |

In **NumPy**, the dimensions are called ***axes***, so this is an array with one axis. We say that this is a 1 x 4 (read “1 by 4”) array; it has 1 row and 4 columns. The ***length*** of the array is the number of elements, which is 4.

A one dimensional column vector might be depicted as:

|  |  |
| --- | --- |
| 0 | 47 |
| 1 | 6 |
| 2 | 15 |

Again, this is an array with one axis. We say that it is a 3 x 1 array; it has 3 rows and 1 column. The length of this array is 3.

Elements in a one dimensional array are found using one index.

A ***two dimensional array*** has two axes. A two dimensional array looks like a table, with both rows and columns. For example, a 2 x 4 array (2 rows, 4 columns) might be depicted as:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 | -2 | 18 | 9 | 32 |
| 1 | 44 | -7 | 4 | 99 |

The first axis is the rows, so the length of the first axis is 2. The second axis is the columns, so the length of the second axis is 4. Two dimensional arrays have indices for both the rows and the columns, and both start at 0. Elements are indexed using two indices, the first for the row, and the second for the column.

In general, an ***n-dimensional array*** has n dimensions, or axes. Elements are indexed using n indices. Arrays with two or more dimensions are frequently called ***matrices***.

A ***scalar***, which is a single value, is said to have a dimension of 0 in NumPy. Scalars are not indexed.

**10.2 Creating Arrays and Subarrays**

**The array function and attributes of arrays:**

The basic function that is used to create arrays is the function **ndarray**, but this function has a simpler alias called **array**, which we will use. One method for creating a one dimensional array is to convert a list (or a tuple) to an array using the **array** function. For the rest of this chapter, we will assume that the **array** function has been imported.

*>>> from numpy import array*

*>>> arr1d = array([31, 42, 11])*

Note how the array is displayed by just typing the variable name and by using the **print** function, and note the type of the array.

*>>> arr1d*

array([ 31, 42, 11])

*>>> print(arr1d, type(arr1d))*

[31 42 11] <class 'numpy.ndarray'>

The type of the data elements can be determined using the **dtype** method.

*>>> print(arr1d.dtype)*

int64

The type **int64** is an integer type that uses 64 bits to store integers.

There are several methods that return the number of dimensions, and the lengths of the axes. The **ndim** method returns the number of dimensions, or in other words the number of axes.

*>>> arr1d.ndim*

1

The lengths of the axes are found using the **shape** method.

*>>> arr1d.shape*

(3,)

In this case, there is only one axis since it is a one-dimensional array, and the length is 3. Notice that the result is returned as a tuple.

The **size** of an array is the number of elements. This is always going to be a single integer.

*>>> arr1d.size*

3

Two dimensional arrays can be created by passing a nested list (a list of lists) to the **array** function. In the following example, two lists, each with 3 numbers, are used to create a 2 x 3 array.

*>>> arr2d = array([[1, 3, 8], [-2, 5, 33]])*

*>>> print(arr2d)*

[[ 1 3 8]

[-2 5 33]]

The two nested lists must have the same number of elements, so there are the same number of elements in each row of the array. The number of dimensions is 2. The **shape** function returns the tuple (2, 3) since it is a 2 x 3 array, and the **size** (the total number of elements) is 6.

*>>> arr2d.ndim*

2

*>>> arr2d.shape*

(2, 3)

*>>> arr2d.size*

6

**Functions and methods that create arrays:**

There are quite a few ways in which arrays can be created. Arrays can be created using a range:

*>>> rgarr = array(range(5))*

*>>> print(rgarr)*

[0 1 2 3 4]

The **numpy** function **arange** specifically creates an array using a range.

*>>> rgarr = np.arange(5)*

*>>> print(rgarr)*

[0 1 2 3 4]

Like the **range** function, **arange** can specify starting, ending, and step values.

*>>> myarr = np.arange(3,9,2)*

*>>> print(myarr)*

[3 5 7]

There are several other functions that can be used to create arrays.

The **linspace** function can be used to create array elements that are linearly spaced within a specified range. The following specifies that the range begins with 1, ends with 9, with 3 linearly spaced points. The **linspace** function determines the step value (in this case, 4). Notice that the default type returned by **linspace** is **float64**, even if the numbers are integers.

*>>> myarr = np.linspace(1,9,3)*

*>>> print(myarr, myarr.dtype)*

[1. 5. 9.] float64

If the number of elements is not specified, the default is 50. Also, the type can be specified using **dtype**.

*>>> myarr = np.linspace(1,9,3,dtype=np.int16)*

*>>> print(myarr, myarr.dtype)*

[1 5 9] int16

The type **int16** is an integer type that uses 16 bits, so it stores smaller integers than the type **int64**.

An array of all zeros can be created using the **zeros** function, by specifying the shape. The default type of each element is **float64**, but like the **linspace** function, this can be modified using **dtype**.

*>>> print(np.zeros((2,4), dtype=int))*

[[0 0 0 0]

[0 0 0 0]]

If only one integer *n* is specified for the shape, a 1 x n row vector is created.

*>>> zarr = np.zeros(3)*

>>> print(zarr)

[0. 0. 0.]

Three dimensional arrays can be created by specifying a tuple of 3 integers for the shape. For example, a shape of (2, 3, 5) creates two 3 x 5 arrays. Notice that when displayed, the two arrays are printed separately with a blank line in between them.

*>>> print(np.zeros((2,3,5), dtype=np.int16))*

[[[0 0 0 0 0]

[0 0 0 0 0]

[0 0 0 0 0]]

[[0 0 0 0 0]

[0 0 0 0 0]

[0 0 0 0 0]]]

Images are often stored as three dimensional arrays. Arrays with more than 3 dimensions can also be created, but it becomes more difficult to visualize them and to find useful applications for them.

One reason for creating an array of all zeros might be if they are going to be running sums or counters.

Arrays of all ones can be created using the **ones** function, which uses the same format as the **zeros** function. An array of all ones might be used as a starting point for running products.

*>>> print(np.ones(6, dtype=int))*

[1 1 1 1 1 1]

To fill in an array with any number other than zeros or ones, the **full** function can be used.

*>>> print(np.full((2,3), 11))*

[[11 11 11]

[11 11 11]]

Instead of just a single number, a set of numbers to be used for each row can be passed to the **full** function, for example using **range**.

*>>> newmat = np.full((3,4), range(4))*

*>>> print(newmat)*

[[0 1 2 3]

[0 1 2 3]

[0 1 2 3]]

Another function that creates arrays is the **empty** function, which contrary to its name does not create an empty array, but fills in the elements with whatever is currently in the memory locations used for the array. This function is useful when the shape of an array is known, but the actual values in the elements will be filled in later. Of course, your results may vary!

*>>> weird = np.empty(4)*

*>>> print(weird)*

[-2.68156159e+154 -3.11108761e+231 2.68678769e+154 2.82470694e-309]

It is frequently useful to create arrays in which random numbers are stored in the elements. **NumPy** has a **random** module that has a method, **default\_rng**, which is used to create a random number generator. Once that has been created, the method **random** can be used to create random floats, and the method **integers** can be used to create random integers in a specified range.

An integer can be passed to **default\_rng** which is the seed for the random number generator; if no argument is passed the seed will be obtained from the operating system.

*>>> rng = np.random.default\_rng()*

*>>> farr = rng.random(3)*

*>>> print(farr)*

[0.74899722 0.9414599 0.70685837]

In the following example, a 1 x 4 array of random integers is created in the range from 3 to 9 inclusive.

*>>> rng.integers(3,10,4)*

array([9, 6, 3, 9])

**Indexing and Slicing:**

Arrays can be indexed and sliced similarly to Python sequences such as lists. The indices begin at 0, and negative indices allow indexing from the end of the array.

*>>> arr1d = array([31, 42, 11])*

*>>> arr1d[1]*

42

*>>> arr1d[-1]*

11

With two dimensional arrays, the row index is given first, and then the column index.

*>>> arr2d = array([[1, 3, 8], [-2, 5, 33]])*

*>>> print(arr2d)*

[[ 1 3 8]

[-2 5 33]]

*>>> arr2d[1,2]*

33

Slicing creates another array.

*>>> arr1d[:]*

array([31, 42, 11])

*>>> arr2d[0,1:]*

array([3, 8])

In addition to using integer indices, **NumPy** allows ***logical indexing***. With logical indexing, Boolean expressions resulting in **True** or **False** can be used to index into an array. This returns the elements from the array for which the corresponding element in the logical array is **True**.

*>>> logarr = arr1d > 20*

*>>> logarr*

array([ True, True, False])

*>>> arr1d[logarr]*

array([31, 42])

Notice that for a two dimensional array, with logical indexing the resulting array is flattened into a one dimensional array. The elements are taken from the original array one row at a time.

*>>> print(arr2d)*

[[ 1 3 8]

[-2 5 33]]

*>>> arr2d[arr2d > 0]*

array([ 1, 3, 8, 5, 33])

**Changing Dimensions:**

There are NumPy methods that change the shape of an array.

The **reshape** method can take any array and reshape it into a new array, as long as the new array has the same number of elements as the original. For example, a 1 x 6 array could be reshaped into a 6 x 1 array, a 2 x 3 array, or a 3 x 2 array.

*>>> myarr = np.arange(6)*

*>>> print(myarr)*

[0 1 2 3 4 5]

*>>> myarr.reshape(6,1)*

array([[0],

[1],

[2],

[3],

[4],

[5]])

*>>> arr2by3 = myarr.reshape(2,3)*

*>>> print(arr2by3)*

[[0 1 2]

[3 4 5]]

The **reshape** method returns a reshaped array, but does not modify the original array. The **resize** method works just like **reshape** except that it does modify the original array.

*>>> myarr = np.arange(6)*

*>>> myarr.resize(3,2)*

*>>> print(myarr)*

[[0 1]

[2 3]

[4 5]]

The **ravel** method flattens an n-dimensional array into a one dimensional array. The default is that it flattens it one row at a time. The **ravel** method returns a flattened array, but does not modify the original.

*>>> print(arr2by3)*

[[0 1 2]

[3 4 5]]

*>>> arr2by3.ravel()*

array([0, 1, 2, 3, 4, 5])

*>>> print(arr2by3)*

[[0 1 2]

[3 4 5]]

The method named simply **T** ***transposes*** an array, which for a two dimensional array means that it interchanges the rows and columns. The method returns a new array, but does not modify the original. In the following example, the first row [0 1 2] becomes the first column, and the second row becomes the second column. The other way to view it is that the first column becomes the first row, the second column becomes the second row, and the third column becomes the third row.

*>>> arr3by2 = arr2by3.T*

*>>> print(arr3by2)*

[[0 3]

[1 4]

[2 5]]

**Combining Arrays:**

There are several functions that create new arrays by combining existing arrays.

The function **vstack** vertically ***stacks***, or ***concatenates***, arrays. A tuple consisting of the arrays to be stacked is passed to the **vstack** function. In this case the one-dimensional arrays have to have the same number of elements so that the columns will match up in the stacked array.

*>>> arrone = np.array([4,11,9])*

*>>> arrtwo = np.array([3,15,22])*

*>>> np.vstack((arrone, arrtwo))*

array([[ 4, 11, 9],

[ 3, 15, 22]])

The function **hstack** horizontally stacks arrays.

*>>> np.hstack((arrtwo,arrone))*

array([ 3, 15, 22, 4, 11, 9])

**10.3 Functions and Operations on Arrays**

NumPy has functions that take arrays as arguments, and operations that can be performed on entire arrays.

Functions

NumPy has ***universal functions*** (***ufuncs***) to which an entire array can be passed. The function is evaluated on every element, returning an array with the same dimensions as the original. These include trig functions such as **sin**, **cos**, **tan**, etc. For example:

from math import pi

x = np.linspace(0, 2\*pi, 10)

y = np.sin(x)

print(x)

print(y)

[0. 0.6981317 1.3962634 2.0943951 2.7925268 3.4906585

4.1887902 4.88692191 5.58505361 6.28318531]

[ 0.00000000e+00 6.42787610e-01 9.84807753e-01 8.66025404e-01

3.42020143e-01 -3.42020143e-01 -8.66025404e-01 -9.84807753e-01

-6.42787610e-01 -2.44929360e-16]

Other ufuncs include the square root function **sqrt**, the absolute value function **abs**, and exponential functions. For example, we could get the absolute value of every element in a 2D array:

rng = np.random.default\_rng()

nums = rng.integers(-10,10, (3,5))

print(nums)

print()

print(np.abs(nums))

[[ 1 -1 7 -10 3]

[ 6 2 6 -5 -3]

[ -1 1 -10 -6 9]]

[[ 1 1 7 10 3]

[ 6 2 6 5 3]

[ 1 1 10 6 9]]

There are ***aggregation functions*** that perform tasks such as summing elements, multiplying elements, and finding minimum and maximum values. By default, the aggregation is performed on the entire array, but to aggregate on just rows or columns, the axis can be specified.

For example, let’s first create a 2D array:

>>> rng = np.random.default\_rng()

>>> nums = rng.integers(60, 100, (4,2))

>>> print(nums)

[[96 91]

[68 64]

[60 97]

[77 77]]

To find the overall minimum, the **min** function is used:

>>> np.min(nums)

60

To find the minimum for each column, specify axis 0:

>>> np.min(nums, axis = 0)

array([60, 64])

To find the minimum for each row, specify axis 1:

>>> smin = np.min(nums, axis = 1)

>>> smin

array([91, 64, 60, 77])

Other unfuncs that work the same way (overall, or specify the axis) include **max**, **mean**, **median**, **std** (standard deviation), and **var** (variance).

The aggregation functions also include **sum** and **prod**. For example, to get an overall sum:

>>> rng = np.random.default\_rng()

>>> nums = rng.integers(0,10, (3,4))

>>> print(nums)

[[3 9 7 3]

[1 0 2 8]

[2 0 0 1]]

>>> np.sum(nums)

36

To get a sum for every column:

>>> np.sum(nums, axis = 0)

array([ 6, 9, 9, 12])

Operations

Numerical operations can be performed on entire arrays. ***Scalar operations*** involve an array and one scalar. For example, to add 5 to every element in an array:

>>> rowvec = array([-3, 28, 6])

>>> rowvec + 5

array([ 2, 33, 11])

Operations such as addition, subtraction, multiplication, division, and exponentiation can be performed on arrays of any dimension. For example, to divide every element in a 2D array by 2:

rng = np.random.default\_rng()

nums = rng.integers(10,20, (2,5))

print(nums)

print()

print(nums/2)

[[18 17 13 11 15]

[14 12 13 12 16]]

[[9. 8.5 6.5 5.5 7.5]

[7. 6. 6.5 6. 8. ]]

***Array operations*** are operations that are performed element by element on two arrays that have the same dimensions.

For example, the following creates two 1 x 5 arrays, or vectors, and adds the corresponding elements to create another 1 x 5 array. This is array addition.

>>> vec1 = array([6, 2, 7, 5, 9])

>>> vec2 = array([3, 1, 4, 2, 1])

>>> vec1 + vec2

array([ 9, 3, 11, 7, 10])

Similarly, array multiplication is an element-by-element operation:

rng = np.random.default\_rng()

mat1 = rng.integers(0,10, (2,3))

print(mat1)

print()

mat2 = rng.integers(0,10, (2,3))

print(mat2)

print()

print(mat1\*mat2)

[[3 9 3]

[4 0 0]]

[[4 2 6]

[8 1 3]]

[[12 18 18]

[32 0 0]]

Note: this is array multiplication, not matrix multiplication. Matrix multiplication is not performed element-by-element, and will be described in the next section.

Matrix Operations and Matrix Properties

***Matrix multiplication*** has a very specific meaning. By “matrix” here we are referring to 2D arrays. First of all, to multiply a matrix A by a matrix B to result in a matrix C, the number of columns of A must be the same as the number of rows of B. If the matrix A has dimensions *m x n*, that means that matrix B must have dimensions *n x* *something*; called *p*.

The terminology is that the ***inner dimensions*** (the *n*s) must be the same. The resulting matrix C has the same number of rows as A and the same number of columns as B (i.e., the ***outer dimensions*** *m x p*). In mathematical notation,

[A]*m x n* [B]*n x p* = [C]*m x p*

This only defines the size of C, not how to find the elements of C.

The elements of the matrix C are defined as the sum of products of corresponding elements in the rows of A and columns of B, or in other words,

cij = .

In the following example, A is *2 x 3* and B is *3 x 4*; the inner dimensions are both 3, so performing the matrix multiplication A\*B is possible (note that B\*A would not be possible). C will have as its size the outer dimensions *2 x 4*. The elements in C are obtained using the summation just described. The first row of C is obtained using the first row of A and in succession the columns of B. For example, C(1,1) is 3\*1+8\*4+0\*0 or 35. C(1,2) is 3\*2+8\*5+0\*2 or 46.

A B C

 \* 

In NumPy, the matrix multiplication operator is @.

>>> A = array([[3, 8, 0],[1, 2, 5]])

>>> B = array([[1, 2, 3, 1], [4, 5, 1, 2], [0, 2, 3, 0]])

>>> C = A @ B

>>> print(C)

[[35 46 17 19]

[ 9 22 20 5]]

Properties of Square Matrices

A ***square matrix*** is a matrix in which the number of rows is the same as the number of columns. The definitions that follow in this section only apply to square matrices.

The ***diagonal*** of a square matrix is the set of terms aii for which the row and column indices are the same, so from the upper left element to the lower right. For example, for the following matrix the diagonal consists of 1, 6, 11, and 16.



A square matrix is a ***diagonal matrix*** if all values that are not on the diagonal are 0. The numbers on the diagonal, however, do not have to be all nonzero, although frequently they are. Mathematically, this is written as aij = 0 for i ~= j. The following is an example of a diagonal matrix:



The NumPy **diag** function can be used to create a diagonal matrix, by passing the numbers to be on the diagonal (either as a 1D array or a list).

>>> diagnums = [4, 5, 2, 7]

>>> np.diag(diagnums)

array([[4, 0, 0, 0],

[0, 5, 0, 0],

[0, 0, 2, 0],

[0, 0, 0, 7]])

If instead a diagonal matrix is passed to the **diag** function, it returns the diagonal as a 1D array.

>>> myd = np.diag([33, 2, 11])

>>> np.diag(myd)

array([33, 2, 11])

So, the **diag** function can be used two ways: (i) pass a matrix and it returns a vector, or (ii) pass a vector and it returns a matrix!

A square matrix is an ***identity*** matrix called I if *aij* = 1 for *i* == *j* and *aij*= 0 for *i* ~= *j*. In other words, all of the numbers on the diagonal are 1 and all others are 0. The following is a *3 x 3* identity matrix:



Note that any identity matrix is a special case of a diagonal matrix.

Identity matrices are very important and useful. NumPy has a built-in function **eye** that will create an *n x n* identity matrix, given the value of *n*:

>>> np.eye(4)

array([[1., 0., 0., 0.],

[0., 1., 0., 0.],

[0., 0., 1., 0.],

[0., 0., 0., 1.]])

A square matrix is ***symmetric*** if *aij = aji* for all *i*, *j*. In other words, all of the values opposite the diagonal from each other must be equal to each other. In this example, there are three pairs of values opposite the diagonals, all of which are equal (the 2s, the 9s, and the 4s).

**10.4 Assigning and Copying Array Variables**

NumPy arrays contain the actual data, and also ***metadata*** such as the data type.

Since array variables are mutable, assigning one array variable to another works like lists. No new object is created.

>>> vec1 = array([6, 2, 7, 5, 9])

>>> vec2 = vec1

>>> vec2 is vec1

True

So, changing one variable changes the other since they are both referring to the same array.

>>> vec1[2] = 11

>>> print(vec1)

>>> print(vec2)

[ 6 2 11 5 9]

[ 6 2 11 5 9]

Assigning a new array to either of the variables creates a completely new location.

>>> vec1 = np.arange(2,6)

>>> print(vec1)

>>> print(vec2)

[2 3 4 5]

[ 6 2 11 5 9]

If it is desired to have two array variables store the same values, but not to refer to the same lo-cation, the **copy** function can be used instead of the assignment operator.

This creates what is called a ***deep copy***, and copies all of the metadata.

>>> vec1 = array([6, 2, 7, 5, 9])

>>> vec2 = np.copy(vec1)

>>> vec2 is vec1

False

**Dictionaries and Intro to Pandas**

Pandas is a data manipulation library that is built on NumPy. The basic Pandas data structures are Series and DataFrames. Series are essentially one-dimensional arrays and DataFrames are two-dimensional arrays, but the row and/or columns can be labeled. Frequently DataFrames are created by reading in data from a csv (comma separated value) file or spreadsheet file, but they can also be created programmatically which is what we will focus on here. Python dictionaries are commonly used to create Series and DataFrames. In this section, we will first introduce dictionaries, and then in an introduction to Pandas, will use dictionaries to create Series and DataFrames.

Dictionaries

***Dictionaries*** are a Python data structure, but they are a ***mapping*** type, not a sequence. This is because indexing into a dictionary is accomplished using ***keys***, not integers. The type of a dictionary is **dict**.

Dictionaries are created by putting ***key-value pairs*** in curly braces. Keys are names specified as strings. The pair is specified as a key followed by a colon, followed by the value for that key, as in ‘key’:value. The following is an example of a dictionary that stores information about a particular hurricane.

*>>> bigone = {'name':'Bertha', 'year': 1952, 'category': 4}*

Values can be retrieved from a dictionary by giving the name of the dictionary variable and then the name of the key in square brackets.

*>>> bigone['year']*

1952

An error will result if the key does not exist in the dictionary.

*>>> bigone['size']*

KeyError: 'size'

An error will also occur if you try to index using an integer; values are only referenced using keys.

Dictionaries are a mutable type. Values can be modified using assignment statements.

*>>> bigone['category'] = 5*

*>>> bigone*

{'name': 'Bertha', 'year': 1952, 'category': 5}

More key-value pairs can be added to a dictionary in the same way.

*>>> bigone['eyewidth'] = 400*

*>>> bigone*

{'name': 'Bertha', 'year': 1952, 'category': 5, 'eyewidth': 400}

Key-value pairs can be deleted from a dictionary using the **del** command.

*>>> del bigone['eyewidth']*

*>>> bigone*

{'name': 'Bertha', 'year': 1952, 'category': 5}

The **len** function returns the number of key-value pairs in the dictionary.

*>>> len(bigone)*

3

The **in** and **not in** operators can be used to find whether a key is in a dictionary or not.

*>>> 'year' in bigone*

True

*>>> 'sqarea' not in bigone*

True

The **list** function will return a list of all of the key names from a dictionary.

*>>> list(bigone)*

['name', 'year', 'category']

There are a number of methods that work with dictionaries.

The **get** method retrieves a value from a dictionary; it returns **None** by default if the key is not in the dictionary.

*>>> bigone.get('year')*

1952

If the key is not in the dictionary, the value **None** that is returned is not automatically shown, but can be printed, or tested in a selection statement.

*>>> print(bigone.get('size'))*

None

A value other than None can also be specified to be used if the key is not in the dictionary, e.g. a flag value of -999.

*>>> bigone.get('size', -999)*

-999

Note that this does not add a key-value pair to the dictionary.

The **get** method is therefore safer to use than just putting a key name in square brackets, since **get** will return a value if the key is not found, whereas using square brackets will result in an error if the key is not found.

The **pop** method deletes an item from a dictionary and returns its value. A default value to return can be specified in case the key is not in the dictionary; if the default is not specified and the key is not found in the dictionary, an error results.

*>>> delcat = bigone.pop('category')*

*>>> delcat*

5

*>>> bigone*

{'name': 'Bertha', 'year': 1952}

*>>> bigone.pop('size')*

KeyError: 'size'

*>>> bigone.pop('size', -999)*

-999

The **popitem** method deletes the last key-value pair in a dictionary, and returns this as a tuple.

*>>> k,v = bigone.popitem()*

*>>> print(k,v)*

year 1952

The **clear** method deletes all of the key-value pairs from a dictionary.

*>>> bigone.clear()*

*>>> bigone*

{}

There are also methods that can be used to iterate: **keys()**, **values()**, and **items()**, for the dictionary’s keys, values, and key-value pairs, respectively.

For example, the following call to the **keys** method returns the names of the keys. Technically, it returns a ***view object***, *dict\_keys*.

*>>> bigone = {'name':'Bertha', 'year': 1952, 'category': 4}*

*>>> bigone.keys()*

dict\_keys(['name', 'year', 'category'])

Normally, the view object would not be accessed or displayed, however. Instead, it would be typical to iterate over the result.

*for kname in bigone.keys():*

*print('The key is', kname)*

The key is name

The key is year

The key is category

To loop over the key-value pairs, the **items** method is used, and two iterator variables are created to store the keys and values individually.

*for k, v in bigone.items():*

*print('The value of',repr(k),'is',v)*

The value of 'name' is Bertha

The value of 'year' is 1952

The value of 'category' is 4

We have stored information about one hurricane in a dictionary. In order to store information on more than one hurricane, we could create a list of dictionaries.

*>>> hurr0 = {'name': 'Bertha', 'year': 1952, 'category': 4}*

*>>> hurr1 = {'name': 'Bob', 'year': 1990, 'category': 2}*

*>>> hurr2 = {'name': 'Camilla', 'year': 1960, 'category': 5}*

*>>> hurricanes = [hurr0, hurr1, hurr2]*

*>>> hurricanes[2] # show example*

{'name': 'Camilla', 'year': 1960, 'category': 5}

Once we have a list of dictionaries, we could iterate through the list, for example to print information on the more powerful hurricanes.

*print('The largest hurricanes were:')*

*for hurr in hurricanes:*

*if hurr['category'] >= 4:*

*print(repr(hurr['name']))*

The largest hurricanes were:

'Bertha'

'Camilla'

It is also possible for values in a dictionary to be data structures themselves. This has already been done, since strings are a sequence data type. Another possibility would be to have a list as a value. Note also in this example that each key-value pair is entered on a separate line, which makes it easier to read.

*icdessert = {*

*'flavor': 'malted',*

*'cone': True,*

*'nscoops': 2,*

*'addins': ['pecans', 'chocolate chips']*

*}*

*icdessert['addins']*

['pecans', 'chocolate chips']

Since the key 'addins' is a list, it can be indexed.

*>>> icdessert['addins'][0]*

'pecans'

We could use a **for** loop to print the elements from the list individually.

*print('Your ice cream add-ins are:')*

*for yummy in icdessert['addins']:*

*print(yummy.title(),'!!!')*

Your ice cream add-ins are:

Pecans !!!

Chocolate Chips !!!

For a dictionary comprehension, key-value pairs must be created.

*>>> cubedict = {i: i\*\*3 for i in range(5)}*

*>>> cubedict*

*{0: 0, 1: 1, 2: 8, 3: 27, 4: 64}*

**Introduction to Pandas**

Since Pandas is built on NumPy, they are frequently imported together using the following:

*>>> import numpy as np*

*>>> import pandas as pd*

The rest of this section will assume that these have been imported.

**Series**

A Pandas Series is a one-dimensional array that can be created using a list or an array. For example,

>>> numseries = pd.Series([11, 33, 15, 2])

>>> numseries

0 11

1 33

2 15

3 2

dtype: int64

Notice that the indices are explicitly listed as part of the Series (which must be capitalized) and that the **dtype** is also stored in the variable and displayed.

The numbers can be accessed using **values**:

>>> numseries.values

array([11, 33, 15, 2])

You can see that it is stored as a NumPy array. You can index into the Series and slice it, just like in Python and NumPy.

>>> numseries[0]

11

>>> numseries[1:3]

1 33

2 15

dtype: int64

The indices can be accessed using **index**, although the result might not be what you would expect!

>>> numseries.index

RangeIndex(start=0, stop=4, step=1)

From this you can see that the indices are a range beginning at 0. Indices can also be specified.

>>> numlabels = pd.Series([11, 33, 15, 2], ['a','b','c','d'])

>>> numlabels

a 11

b 33

c 15

d 2

dtype: int64

The labels can then be used to index and slice into the Series.

>>> numlabels['c']

15

>>> numlabels['b':'d']

b 33

c 15

d 2

dtype: int64

Implicit integer indices can also be used to index and to slice into the Series. Notice, however, that using 'b':'d' returns 3 rows, whereas [1:3] only returns 2 (like Python and NumPy).

>>> numlabels[1:3]

b 33

c 15

dtype: int64

A Series can also be created using a dictionary.

>>> numdict = {'a':11, 'b':33, 'c':15, 'd':2}

>>> numlab = pd.Series(numdict)

>>> numlab

a 11

b 33

c 15

d 2

dtype: int64

**DataFrames**

A Pandas DataFrame looks like a two-dimensional array with labels for the rows and the columns. A DataFrame can be constructed from a single Series, for example from above:

>>> nl = pd.DataFrame(numlab,columns = ['Nums'])

>>> nl

|  | **Nums** |
| --- | --- |
| **a** | 11 |
| **b** | 33 |
| **c** | 15 |
| **d** | 2 |

DataFrames usually have multiple columns, however.

Let’s create a DataFrame with information on students by constructing multiple Series using dictionaries.

id\_dict = {'Joey': 123, 'Javier': 234, 'Juanita': 456, 'Jane': 678}

ids = pd.Series(id\_dict)

exam1\_dict = {'Joey': 95, 'Javier': 99, 'Juanita': 88, 'Jane': 100}

exam1s = pd.Series(exam1\_dict)

students = pd.DataFrame({'ID': ids, 'Exam1': exam1s})

students

|  | **ID** | **Exam1** |
| --- | --- | --- |
| **Joey** | 123 | 95 |
| **Javier** | 234 | 99 |
| **Juanita** | 456 | 88 |
| **Jane** | 678 | 100 |

The **index** attribute shows the indices, which are in an Index object that stores the row labels.

>>> students.index

Index(['Joey', 'Javier', 'Juanita', 'Jane'], dtype='object')

The **columns** attribute shows the indices, which are in an Index object that stores the column labels.

>>> students.columns

Index(['ID', 'Exam1'], dtype='object')

The **values** attribute returns the numbers in the two columns as a 2D array.

>>> students.values

array([[123, 95],

[234, 99],

[456, 88],

[678, 100]])

In order to access a column in the DataFrame, the column name can be specified in two ways:

>>> idcol = students['ID']

>>> idcol

Joey 123

Javier 234

Juanita 456

Jane 678

Name: ID, dtype: int64

>>> idcol = students.ID

>>> idcol

Joey 123

Javier 234

Juanita 456

Jane 678

Name: ID, dtype: int64

Note that this is a Series. Then, to access an individual Id from the Series, we can index using the implicit integer index or using the row label.

>>> idcol[0]

123

>>> idcol['Javier']

234

Of course, it is not necessary to extract the column first before indexing to get an individual value. These can be combined.

>>> students.ID[0]

123

>>> students['ID']['Jane']

678

In order to access a row in a DataFrame, the **iloc** method can be used with an implicit index, or the **loc** method can be used with a label.

>>> students.loc['Juanita']

ID 456

Exam1 88

Name: Juanita, dtype: int64

>>> students.iloc[2]

ID 456

Exam1 88

Name: Juanita, dtype: int64

Note that this is a Series.

There are several functions and methods that can be used to find the dimensions of a DataFrame. The **len** function can be used to find the number of rows, and by specifying .columns the number of columns.

>>> len(students)

4

>>> len(students.columns)

2

The **shape** method will return both dimensions as a tuple.

>>> students.shape

(4, 2)

A column can be added to the DataFrame as follows:

>>> students['Exam2'] = [100, 99, 98, 97]

>>> students

|  | **ID** | **Exam1** | **Exam2** |
| --- | --- | --- | --- |
| **Joey** | 123 | 95 | 100 |
| **Javier** | 234 | 99 | 99 |
| **Juanita** | 456 | 88 | 98 |
| **Jane** | 678 | 100 | 97 |

Column(s) can be deleted using the **drop** method. This must be assigned to the DataFrame object.

>>> students = students.drop(columns = 'ID')

>>> students

|  | **Exam1** | **Exam2** |
| --- | --- | --- |
| **Joey** | 95 | 100 |
| **Javier** | 99 | 99 |
| **Juanita** | 88 | 98 |
| **Jane** | 100 | 97 |

Statistical methods that can be used include mean(), sum(), min(), max(), std(), and var(). They can be used with the entire DataFrame, or an individual column.

>>> students.mean()

Exam1 95.5

Exam2 98.5

dtype: float64

If a DataFrame object is large, which it frequently will be if read from a file, the first 5 rows can be viewed using the **head**() method, and the last 5 rows can be viewed using the **tail**() method.

**Scalar operations**

Scalar operations can be performed on arrays (vectors or matrices). For example, one can perform scalar operations on the following vector *vec*:

>> vec = 1: 0.5: 3

vec =

1.0000 1.5000 2.0000 2.5000 3.0000

Multiplying the vector by 2 performs the multiplication on every element in the vector, resulting in a vector with the same length as the original. Since this is not stored in a variable, the result is stored in the default variable *ans*.

>> vec \* 2

ans =

2 3 4 5 6

All numerical operators can be used in this way. For example, we can add 5 to every element:

>> newv = vec + 5

newv =

6.0000 6.5000 7.0000 7.5000 8.0000

Scalar operations can also be performed on matrices.

>> mat = randi([5, 20], 2, 3)

mat =

17 15 18

20 5 19

>> mat / 2

ans =

8.5000 7.5000 9.0000

10.0000 2.5000 9.5000

**Array operations**

Array operations are operations that are performed on corresponding elements of two arrays, so they must have the same dimensions.

>> vec1 = [2 11 33 5];

>> vec2 = linspace(3, 9, 4)

vec2 =

3 5 7 9

>> vec1 + vec2

ans =

5 16 40 14

For operations that are based on multiplication, the operator must have a dot in front of it. So, the multiplication array operators are:

.^

.\*

./

.\

>> vec2

vec2 =

3 5 7 9

>> vec2 .^ 2

ans =

9 25 49 81

>> mata = [1 3; 2 1]

mata =

1 3

2 1

>> matb = [4 2; 1 5]

matb =

4 2

1 5

>> mata .\* matb

ans =

4 6

2 5

**Matrix multiplication**

So, .\* is array multiplication. The matrix multiplication operator is just \*.

>> matc = mata \* matb

matc =

7 17

9 9

**Logical operators**

Logical operators can be used with arrays, also.

>> vec = randi([1,20], 1,5)

vec =

16 15 8 14 4

>> isless10 = vec < 10

isless10 =

1x5 [logical](matlab:helpPopup('logical')) array

0 0 1 0 1

Logical indexing also works in MATLAB.

>> vec(isless10)

ans =

8 4

>> vec == vec

ans =

1x5 [logical](matlab:helpPopup('logical')) array

1 1 1 1 1

To find out just true/false whether two vectors are equal to each other, use **isequal**.

>> isequal(vec, vec)

ans =

[logical](matlab:helpPopup('logical'))

1

**Arrays as Function Arguments**

Entire arrays can be passed to functions. The function will evaluate each of the elements in the array and return an array with the same dimensions as the original.

>> myvec = [-3: 2]

myvec =

-3 -2 -1 0 1 2

>> abs(myvec)

ans =

3 2 1 0 1 2

>> mymat = randi([0, 5], 2, 4)

mymat =

0 5 3 4

2 2 1 1

>> sqrt(mymat)

ans =

0 2.2361 1.7321 2.0000

1.4142 1.4142 1.0000 1.0000

>> x = linspace(-2\*pi, 2\*pi, 6);

>> y = sin(x)

y =

0.0000 0.5878 -0.9511 0.9511 -0.5878 -0.0000

**Functions that change array dimensions**

The **reshape** function can change the dimensions of any array to any other array that has the same number of elements.

>> ranmat = randi([1,20], 2,4)

ranmat =

9 16 14 17

19 20 1 19

>> reshape(ranmat,4,2)

ans =

9 14

19 1

16 17

20 19

The **rot90** function rotates a matrix 90 degrees counterclockwise.

>> ranmat

ranmat =

9 16 14 17

19 20 1 19

>> rot90(ranmat)

ans =

17 19

14 1

16 20

9 19

There are several functions that flip matrices. The **fliplr** function flips a row vector or the columns of a matrix from left to right.

>> vec = 2:3:14

vec =

2 5 8 11 14

>> fliplr(vec)

ans =

14 11 8 5 2

The **flipud** function flips a column vector or the rows of a matrix up to down.

>> vec = 2:3:14

vec =

2 5 8 11 14

>> fliplr(vec)

ans =

14 11 8 5 2

The **flip** function flips a row vector left to right or a column vector or matrix up to down.

There are functions that replicate a matrix or element from a matrix. The **repmat** function replicates an entire matrix. The following replicates a matrix variable *mymat* 6 times, as a 2 x 3 matrix of mymats.

>> mymat = [1 2; 3 4]

mymat =

1 2

3 4

>> repmat(mymat,2,3)

ans =

1 2 1 2 1 2

3 4 3 4 3 4

1 2 1 2 1 2

3 4 3 4 3 4

The **repelem** function replicates each element, in this case as a 2 x 3 matrix of each element.

>> mymat = [1 2; 3 4]

mymat =

1 2

3 4

>> repelem(mymat, 2,3)

ans =

1 1 1 2 2 2

1 1 1 2 2 2

3 3 3 4 4 4

3 3 3 4 4 4

**Array Functions and Statistical Functions**

There are functions that perform statistical analyses on vectors, including:

* **min**: minimum value
* **max**: maximum value
* **mean**: average
* **mode**: number that appears most frequently
* **median**: number in the middle of a sorted vector
* **std**: standard deviation
* **var**: variance
* **sum**: sum

For example,

>> vec = [5 33 11 2 7 9 4]

vec =

5 33 11 2 7 9 4

>> min(vec)

ans =

2

>> mean(vec)

ans =

10.1429

For matrices, all of these functions work column-wise:

>> mat = randi([1, 10], 3, 5)

mat =

4 4 7 7 1

2 6 3 8 3

8 2 7 5 10

>> max(mat)

ans =

8 6 7 8 10

>> sum(mat)

ans =

14 12 17 20 14

So, notice that the result is a row vector that contains the result of the function for each column. To get an overall, for example overall maximum, get the max of this row vector:

>> max(max(mat))

ans =

10

>> sum(sum(mat))

ans =

77

Other similar functions include:

* **prod**: product
* **cumsum**: cumulative sum
* **cumprod**: cumulative product
* **cummin**: cumulative minimum
* **cummax**: cumulative maximum

These can similarly work on a vector or the columns of a matrix.

>> rvec = 2:5

rvec =

2 3 4 5

>> prod(rvec)

ans =

120

>> rvec = [rvec 11]

rvec =

2 3 4 5 11

>> cumsum(rvec)

ans =

2 5 9 14 25

>> mat = randi([0,5], 3, 4)

mat =

0 4 3 1

1 3 1 4

3 2 4 1

>> prod(mat)

ans =

0 24 12 4

>> cummin(mat)

ans =

0 4 3 1

0 3 1 1

0 2 1 1

The **diff** function returns differences between consecutive elements in a vector.

>> vec = [5 33 11 2 7]

vec =

5 33 11 2 7

>> diff(vec)

ans =

28 -22 -9 5

Note that the result has one fewer element than the original vector.

The **sort** function sorts a vector, or columns of a matrix.

>> mat

mat =

0 4 3 1

1 3 1 4

3 2 4 1

>> sort(mat)

ans =

0 2 1 1

1 3 3 1

3 4 4 4

**Square Matrices**

There are functions that operate on square matrices. The **diag** function returns the diagonal of a matrix, or creates a diagonal matrix by putting a vector on the diagonal.

>> mydm = diag([2:4])

mydm =

2 0 0

0 3 0

0 0 4

>> diag(mydm)

ans =

2

3

4

The trace of a square matrix is the sum of the diagonal; the **trace** function will return this.

>> trace(mydm)

ans =

9

The **eye** function creates an Identity matrix.

>> eye(4)

ans =

1 0 0 0

0 1 0 0

0 0 1 0

0 0 0 1

There are “is” functions that ask True/False questions about square matrices. The **isdiag** function returns 1 for true if a matrix is a diagonal matrix.

>> isdiag(mydm)

ans =

logical

1

The **issymmetric** function returns logical 1 if the matrix is a symmetric matrix.

>> smat = [1 2 3; 2 7 4; 3 4 5]

smat =

1 2 3

2 7 4

3 4 5

>> issymmetric(smat)

ans =

logical

1

**Input**

Input into a MATLAB program is achieved using the **input** function. A character vector, which is the prompt, is passed, and the user’s input should be stored in a variable. For example, to read in a number:

>> favnum = input('Please enter your fave num: ')

Please enter your fave num: 33

favnum =

33

The user could also be prompted for a vector, but would have to know the proper syntax.

>> urvec = input('Enter a vector: ')

Enter a vector: [4 8 2 9]

urvec =

4 8 2 9

>> urvec = input('Enter a vector: ')

Enter a vector: 4:5:25

urvec =

4 9 14 19 24

If the user is to enter a character or a character vector, a second argument is passed to the **input** function, 's'.

>> favpunct = input('Enter your fave punctuation mark: ', 's')

Enter your fave punctuation mark: !

favpunct =

'!'

>> icolor = input('What is your eye color: ', 's')

What is your eye color: brown

icolor =

'brown'

It is not possible to read directly into a string variable. However, a character vector can be converted to a string.

>> stricolor = string(icolor)

stricolor =

"brown"

The **input** function can only read in one thing at a time. Multiple desired inputs means multiple calls to the **input** function. (Note that a vector is considered one thing.)

**Output**

MATLAB has two basic output functions: **disp** and **fprintf**. The **disp** function is a quick way of displaying an expression, but it does not allow any formatting. It does, however, always print a newline character at the end. The **fprintf** function allows custom formatting of the output. Vectors and matrices are best displayed using **disp**.

Here are some examples using **disp**:

>> disp(4e2)

400

>> disp('Hi there!')

Hi there!

>> disp("Hello")

Hello

>> disp(3:7)

3 4 5 6 7

>> disp(eye(3))

1 0 0

0 1 0

0 0 1

The **fprintf** function allows formatting. For example, the following prints the number 33 in a sentence. Two arguments are passed to the **fprintf** function: a ***format string***, and the number 33. The %d in the format string is a ***place holder***, meaning that whatever comes after the format string fills in in that location. The newline character \n at the end moves the cursor down.

>> fprintf('The number is %d.\n', 33)

The number is 33.

>>

Without the newline character, the next prompt would end up on the same line as what was printed. It’s still a prompt, but it looks messy!

>> fprintf('The number is %d.', 33)

The number is 33.>> 5 - 2

ans =

3

The basic place holders are %d for integers (decimal integers), %f for floats, %c for single characters, and %s for character vectors or strings. The format string can be either a character vector or a string.

>> fprintf("The character is '%c'!!\n", 'x')

The character is 'x'!!

>> fprintf("The word is %s!!\n", 'fun')

The word is fun!!

>> fprintf('The word is "%s"!!\n', "fun")

The word is "fun"!!

>> fprintf('The number is %f, I think.\n', 11.11)

The number is 11.110000, I think.

The field width, number of decimal places (for floats) and justification can be specified.

The field width can be specified by putting an integer in between the % and the character in the place holder. For example, to print an integer in a field width of 5:

>> fprintf('The number is %5d!!\n', 33)

The number is 33!!

A negative integer will left-justify instead of right.

>> fprintf('The number is %-5d!!\n', 33)

The number is 33 !!

For floats, the number of decimal places can be specified. For example, a place holder of %6.2f prints in a field width of 6 altogether, including the decimal point and 2 decimal places, so in the format xxx.xx.

>> fprintf('The real number is %6.2f!!\n', 12.3)

The real number is 12.30!!

The field width does not need to be specified. If just the number of decimal places is specified, the field width will be set according to how wide the actual number is. For example,

>> fprintf('The cost was $%.2f, wow.\n', 123.456)

The cost was $123.46, wow.

If more than one expression is to be printed, there will be multiple place holders in the format string.

>> fprintf('Int: %d, Char: %c\n', 33, 'x')

Int: 33, Char: x

**Scripts**

We have seen that commands, statements, and expressions can be entered interactively one at a time in the Command Window. It is also possible to group them together into a script, and then have MATLAB execute the statements in the script sequentially. From the Home tab, you can click on the “New Script” icon, which will bring up a new Editor window. Once the statements have been entered in the Editor, from the Editor tab, you can click on Save to save the file. Code files including simple scripts are stored in files with the extension of .m on the name.

For example, the following was entered in the Editor, and saved in a file named ‘myscript1.m’.

myscript1.m

% This script calculates and prints the area of a rectangle

disp('Enter the length and width of a rectangle in inches.')

rlength = input('Enter the length: ');

rwidth = input('Enter the width: ');

fprintf('The area is %.2f\n', rlength \* rwidth)

The first line in the file is a comment. Comments are anything from a % to the end of that line.

To execute the script, type the name of the code file (without the .m) at the prompt in the Command Window.

>> myscript1

Enter the length and width of a rectangle in inches.

Enter the length: 4.2

Enter the width: 3.3

The area is 13.86

To view the script from the Command Window, the **type** command can be used.

>> type myscript1

% This script calculates and prints the area of a rectangle

disp('Enter the length and width of a rectangle in inches.')

rlength = input('Enter the length: ');

rwidth = input('Enter the width: ');

fprintf('The area is %.2f\n', rlength \* rwidth)

**Selection Statements**

There are several selection statements in MATLAB, including the **if** statement, **if-else** statement (with optional **elseif** clause), and the **switch** statement.

The **if** statement chooses whether an action is executed or not. The general form is:

if condition

action

end

The condition is a Boolean expression that is either true or false. If the condition is true, the action is executed – otherwise it is not executed. The action can be any number of statements up to the key word **end**. The action should be indented to make it easy to see.

For example, if a number should be positive, an if statement would change it if it is negative.

if num < 0

num = abs(num);

end

The **if-else** statement chooses between two actions. The general form is:

if condition

action1

else

action2

end

Again, the condition is a Boolean expression. If the value of the condition is true, the first action (action1) is executed. If instead the condition is false, the second action (action2) is executed. The actions are naturally bracketed by the reserved words **else** and **end**. One, and only one, of the actions will be executed. Which one depends on the value of the condition.

For example,

rint = randi([1, 10]);

if rint <= 5

fprintf('Lower half\n')

else

fprintf('Upper half\n')

end

If multiple actions are desired based on multiple conditions, nested **if-else** statements can be used with the optional **elseif** clause. The general form is:

if condition1

action1

elseif condition2

action2

elseif condition3

action3

else % can have many more elseif

actionn

end

The **switch** statement is used when you want to test a variable to see whether it is equal to different values, with different actions for each. For example, if you have a variable x and want to see whether it is equal to a, b, or c, the code could be written using an **if-else** statement, or a **switch** statement. The algorithms would be:

if x == a

action1

elseif x == b

action2

elseif x == c

action3

else

action4

end

switch x

case a

action1

case b

action2

case c

action3

otherwise

action4

end

The **switch** statement has **case** labels; the value of the variable is compared to the values on the case labels. If none of them are equal, the **otherwise** clause handles “none of the above”.

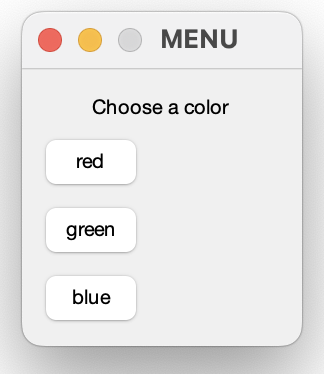
The **switch** statement does not add any power to the language, but may be easier to read.

One application of the **switch** statement is to choose different actions from a menu. MATLAB has a **menu** function that graphically presents the user with options.

For example, the following **menu** function asks the user to choose a color.

>> choice = menu('Choose a color', 'red', 'green', 'blue')

This brings up a box in that looks like this:



The first character vector passed to menu is a prompt that appears at the top. The others are labels that go on buttons. If the user pushes the first button (labeled ‘red’), the value returned from the **menu** function is 1, if the user pushes the second button the value returned is 2, if the user pushes the third button the value is 3, and if the user pushes the red circle, the value returned is 0. Once the value is returned and stored in the variable choice, a **switch** statement could then do different actions based on the user’s choice.

choice = menu('Choose a color', 'red', 'green', 'blue')

switch(choice)

case 1

fprintf('Your chose red!')

case 2

fprintf('You chose green!')

case 3

fprintf('You chose blue!')

otherwise

fprintf('Sadly, you made no choice.')

end

Instead of the **otherwise** clause, in this example a case label of 0 could be used.

**Simple Plots**

There are many simple plot types in MATLAB. Creating and annotating a plot is generally an iterative process, until the plot looks the way you want it to. For this reason, it is usually easiest to write a script to create a plot rather than doing it one line at a time in the Command Window.

To begin, create vectors storing the data points. Plots can be created programmatically, as in the following. This creates an x vector ranging from -pi to 2pi, using **linspace** which will by default create 100 elements. The y vector is sin(x). Both assignments are suppressed. Then, the plot function plots the vectors as points using the color blue and the marker ‘\*’. The plot is then annotated with labels on the x and y axes and a title on top.

x = linspace(-pi, 2\*pi);

y = sin(x);

plot(x,y, 'b\*')

xlabel('x')

ylabel('sin(x)')

title('sin(x) from -pi to 2pi')

This brings up a Figure Window that contains the following plot.



The plot can be modified from the Figure Window, for example under Insert you can insert x and y axis labels and a title directly on the plot.

It is also possible to create plots without using the functions. You can choose variables in the Workspace Window, and then under the Plots tab you can click on a plot type.

**Loops**

In general in programming, there are two kinds of loops: counted loops, that repeat an action a specified number of times, and conditional loops, that repeat an action until something happens or as long as a condition is true. In MATLAB, the **for** loop is the counted loop, and **while** is the conditional loop.

**For loop:**

The general form of a **for** loop is

for loopvar = range

action

end

where *loopvar* is the loop variable, *range* is the range through which it iterates, and action is the set of statements that will be repeated. Frequently the colon operator is used to specify the range. For example, a loop to repeat 5 times would have the form:

for i = 1:5

action

end

This specifies that the loop variable *i* iterates through the integers from 1 to 5, so the action is repeated 5 times.

Sometimes the loop variable is used in the action, and sometimes it is not (it is just used to specify how many times to repeat the action). For example, the following uses the value of i:

for i = 3:-1 :1

disp(i)

end

3

2

1

The following does not use the loop variable in the action:

for i = 1:4

fprintf('!')

end

!!!!

The loop variable can be used to index into a vector variable.

vec = [33 11 2 5];

for i = 1:length(vec)

fprintf('%d: %d\n', i, vec(i))

end

1: 33

2: 11

3: 2

4: 5

If a vector is to be created in a loop, it is best practice to ***preallocate*** the vector first if the eventual length of the vector is known. Vectors can be extended, but that is very inefficient. It is common to preallocate a vector to all zeros. For example:

>> vec = zeros(1,3);

>> for i = 1:3

vec(i) = input('Enter a number: ');

end

Enter a number: 4

Enter a number: 9

Enter a number: 2

>> vec

vec =

4 9 2

**While loops**

The conditional loop in MATLAB is the **while** loop. The general form is:

while condition

action

end

The condition is a Boolean expression. The action is repeated as long as the condition is true. The indentation of the action is not necessary, but makes it easier to read. The action is any number of statements up to the reserved word **end**.

For example, the following loops to prompt the user for a number greater than 50, until the user does this.

num = input('Enter a number > 50: ');

while num <= 50

num = input('Enter a number > 50: ');

end

fprintf('Thanks, you entered %.1f\n', num)

Enter a number > 50: 33

Enter a number > 50: -8

Enter a number > 50: 52

Thanks, you entered 52.0

**Nested loops**

A nested loop is one loop inside the action of another loop. For example, a nested **for** loop to print might look like this:

for i = 1:3

for j = 1:5

fprintf('\*')

end

fprintf('\n')

end

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

In the following example, the script prompts the user for 3 negative numbers to store in a vector, each time looping to error-check until the user enters a negative number.

createnvec.m

negvec = zeros(1,3);

for i = 1:3

num = input('Enter a negative number: ');

while num >= 0

num = input('Enter a negative number: ');

end

negvec(i) = num;

end

disp(negvec)

Here is an example of executing the script:

>> createnvec

Enter a negative number: -4

Enter a negative number: 33

Enter a negative number: 2

Enter a negative number: -7

Enter a negative number: -11

-4 -7 -11

Nested loops can also be used to perform an operation on every element in a matrix. One loop would be over the rows of the matrix, and the other loop over the columns.

For example, given a matrix variable *mat*, we will write code to find the overall sum of the numbers in the matrix, the sum of each row, and the sum of each column (without using the **sum** function).

matsums.m

mat = [1 3 5; 2 6 3]

[r c] = size(mat);

% Calculate the overall sum

mysum = 0;

for i = 1:r

for j = 1:c

mysum = mysum + mat(i,j);

end

end

fprintf('The overall sum is %d.\n\n', mysum)

% Calculate the sum for each row

for i = 1:r

mysum = 0;

for j = 1:c

mysum = mysum + mat(i,j);

end

fprintf('Sum for row %d: %d\n', i, mysum)

end

fprintf('\n')

% Calculate the sum for each column

for j = 1:c

mysum = 0;

for i = 1:r

mysum = mysum + mat(i,j);

end

fprintf('Sum for column %d: %d\n', j, mysum)

end

Executing this would result in:

>> matsums

mat =

1 3 5

2 6 3

The overall sum is 20.

Sum for row 1: 9

Sum for row 2: 11

Sum for column 1: 3

Sum for column 2: 9

Sum for column 3: 8

There are several things to note in this code:

* When a result is calculated for every row, the outer loop has to be over the rows
* When a result is calculated for every column, the outer loop has to be over the columns
* When an overall result is calculated, it does not matter whether the outer loop is over the rows or columns
* The matrix is always indexed with the row index first, then the column index, regardless of the order of the loops
* For an overall sum, the running sum variable is initialized to 0 before the nested loop
* For a sum of every row, the running sum variable must be initialized to 0 for every row
* For a sum of every column, the running sum variable must be initialized to 0 for every column

**Vectorizing Code**

Since MATLAB is written to work with vectors and matrices, it is never necessary to use loops when performing an operation on every element in a vector or matrix, or to call a function on every element in a vector or matrix.

For example, to add 5 to every element in a vector we could loop as follows:

>> vec = [4 9 2];

>> for i = 1:length(vec)

vec(i) = vec(i) + 5;

end

>> vec

vec =

9 14 7

The vectorized code would be instead:

>> vec = [4 9 2];

>> vec = vec + 5

vec =

9 14 7

Similarly, to multiply every element in a matrix by 2, we could use a nested loop as follows:

>> mat = [3:5; 2 11 16]

mat =

3 4 5

2 11 16

>> [r c] = size(mat);

>> for i = 1:r

for j = 1:c

mat(i,j) = mat(i,j) \* 2;

end

end

>> mat

mat =

6 8 10

4 22 32

The vectorized code is:

>> mat = [3:5; 2 11 16]

mat =

3 4 5

2 11 16

>> mat = mat \* 2

mat =

6 8 10

4 22 32

**User-Defined functions**

We will categorize functions as follows:

* Functions that calculate and return one value
* Functions that calculate and return more than one value
* Functions that just accomplish a task, such as printing, without returning any values

Thus, although many functions calculate and return values, some do not. Instead, some functions just accomplish a task. There are differences between these three types of functions, including the format of the function headers and also the way in which the functions are called. Regardless of what kind of function it is, all functions must be defined, and all function definitions consist of the ***header*** and the ***body***. Also, the function must be called for it to be utilized. Although functions can be stored in script code files, for now we will concentrate on functions that are stored in their own code files with an extension of .m.

In general, any function in MATLAB consists of the following:

1. The function header (the first line); this has:
   1. the reserved word **function**
   2. if the function ***returns*** values, the name(s) of the output argument(s), followed by the assignment operator (=)
   3. the name of the function (important: this should be the same as the name of the file in which this function is stored to avoid confusion)
   4. the input arguments in parentheses, if there are any (separated by commas if there is more than one).
2. A comment that describes what the function does (this is printed if **help** is used).
3. The body of the function, which includes all statements, including putting values in all output arguments if there are any.
4. **end** at the end of the function.

### User-defined functions that return a single value

The general form of a ***function definition*** for a function that calculates and returns one value looks like this:

functionname.m

function outputargument = functionname(input arguments)

% Comment describing the function

Statements here; these must include putting a value in the output argument

end % of the function

For example, the following is a function called *calcarea* that calculates and returns the area of a circle; it is stored in a file called *calcarea.m*.

calcarea.m

function area = calcarea(rad)

% calcarea calculates the area of a circle

% Format of call: calcarea(radius)

% Returns the area

area = pi \* rad \* rad;

end

A radius of a circle is passed to the function to the input argument *rad*; the function calculates the area of this circle and stores it in the output argument *area*.

In the function header, we have the reserved word **function**, then the output argument *area* followed by the assignment operator =, then the name of the function (the same as the name of the file), and then the input argument *rad*, which is the radius. As there is an output argument in the function header, somewhere in the body of the function we must put a value in this output argument. This is how a value is returned from the function. In this case, the function is simple and all we have to do is assign to the output argument *area* the value of the built-in constant **pi** multiplied by the square of the input argument *rad*.

The function can be displayed in the Command Window using the **type** command.

>> type calcarea

function area = calcarea(rad)

% calcarea calculates the area of a circle

% Format of call: calcarea(radius)

% Returns the area

area = pi \* rad \* rad;

end

The following is an example of a call to this function in which the value returned is stored in the default variable *ans*:

*>> calcarea(4)*

ans =

50.2655

**Functions that return more than one value**

The general form of a function definition for a function that calculates and returns more than one value looks like this:

functionname.m

function [output arguments] = functionname(input arguments)

% Comment describing the function

% Format of function call

Statements here; these must include putting values in all of the output arguments listed in the header

end

In the vector of output arguments, the output argument names are by convention separated by commas.

Choosing New, then Function brings up a template in the Editor that can then be filled in. If this is not desired, it may be easier to start with New Script.

For example, here is a function that calculates two values, both the area and the circumference of a circle; this is stored in a file called *areacirc.m*:

areacirc.m

function [area, circum] = areacirc(rad)

% areacirc returns the area and

% the circumference of a circle

% Format: areacirc(radius)

area = pi \* rad .\* rad;

circum = 2 \* pi \* rad;

end

As this function is calculating two values, there are two output arguments in the function header (*area* and *circum*), which are placed in square brackets [ ]. Therefore, somewhere in the body of the function, values have to be stored in both.

As the function is returning two values, it is important to capture and store these values in separate variables when the function is called. In this case, the first value returned, the area of the circle, is stored in a variable *a* and the second value returned is stored in a variable *c*:

*>> [a, c] = areacirc(4)*

a =

50.2655

c =

25.1327

If this is not done, only the first value returned is retained - in this case, the area:

*>> disp(areacirc(4))*

50.2655

The **help** function shows the comment listed under the function header:

*>> help areacirc*

This function calculates the area and

the circumference of a circle

Format: areacirc(radius)

**Functions that do not return**

The general form of a function definition for a function that does not return any values looks like this:

functionname.m

function functionname(input arguments)

% Comment describing the function

Statements here

end

Note what is missing in the function header: there are no output arguments and no assignment operator.

For example, the following function just prints the two arguments, numbers, passed to it in a sentence format:

printem.m

function printem(a,b)

% printem prints two numbers in a sentence format

% Format: printem(num1, num2)

fprintf('The first number is %.1f and the second is %.1f\n',a,b)

end

As this function performs no calculations, there are no output arguments in the function header and no assignment operator (=). An example of a call to the *printem* function is:

*>> printem(3.3, 2)*

The first number is 3.3 and the second is 2.0

Note that as the function does not return a value, it cannot be called from an assignment statement. Any attempt to do this would result in an error, such as the following:

>> x = printem(3, 5) % Error!!

Error using printem

Too many output arguments.

We can therefore think of the call to a function that does not return values as a statement by itself, in that the function call cannot be imbedded in another statement such as an assignment statement or a statement that prints.