Begin to code with Python
Obtaining MTA qualification expanded notes

The Microsoft Certified Professional program lets you obtain recognition for your skills. Passing the exam 98-381, "Introduction to Programming Using Python" gives you a Microsoft Technology Associate (MTA) level qualification in Python programming. You can find out more about this examination at https://www.microsoft.com/en-us/learning/exam-98-381.aspx.

To help you get the best out of this text, if you're thinking of using it as part of a program of study to prepare for this exam, I've put together a mapping of exam topics to book elements that you may find helpful, along with some extra notes that put the exam skills into context. (Note that these mappings are based on the exam specification as of October 2017.)

I've also added some extra text to put the Python features explained in the text into the context of the skills described in the test specification.

Important note

The test does not just check to see if you *know* about a particular Python element, it also tests to discover whether you *understand* how that element is used in a Python solution. It is important that you work through all the exercises and examples in the book to make sure that you have a proper understanding of how to use the language.

For example; you could memorize the fact that Python provides integer, floating-point, string and bool data types for variables. However, during the test you will not just be asked "What types of data does Python support?". You may also be required to analyze code to determine the type of variables that are in use in particular statements and may also be asked you to suggest storage types for particular kinds of data.

To answer these questions there is no substitute for working with the language to solve problems.

However, I hope you find the following mapping of qualification elements to book chapters useful.

Qualification structure

The qualification is broken down into a number of topic areas, each of which comprises a percentage of the total qualification. We can go through each topic area and identify the elements of this book that apply. I've added some extra text to put the skills into context and then given references to the parts of the book where these are described.

Perform operations using data types and operators (20-25%)

To prepare for this topic, you should pay special attention to Chapter 4, which describes the essentials of data processing in Python programs as well as how text and numeric data is stored and manipulated. Chapter 5 introduces the Boolean variable type and explains how logical values are used in a program. Chapter 8 describes how to store collections of data, and Chapter 9 explains the creation and storage of data structures. Chapter 11 gives details of sets in Python programs.

Skill	Book Element
Evaluate an expression to identify the data type Python will assign to each variable. Identify str, int, float, and bool data types.	Chapter 4: Variables in Python Chapter 4: Working with text Chapter 4: Working with numbers Chapter 5: Boolean data Chapter 5: Boolean expressions
Perform data and data type operations. Convert from one data type to another type; construct data structures; perform indexing and slicing operations.	Chapter 4: Convert strings into floating-point values Chapter 4: Convert between float and int Chapter 4: Real numbers and floating-point numbers Chapter 8: Lists and tracking sales
Determine the sequence of execution based on operator precedence = Assignment; Comparison; Logical; Arithmetic; Identity (is); Containment (in)	Chapter 4: Performing calculations Chapter 9: Contact objects and references Chapter 11: Sets and tags
Select the appropriate operator to achieve the intended result: Assignment; Comparison; Logical; Arithmetic; Identity (is); Containment (in).	Chapter 4: Performing calculations Chapter 5: Boolean expressions

Here's a breakdown of skills which add some more detail and refers to the specific book pages that will deal with this skill:

Evaluate an expression to identify the data type Python will assign to each variable. Identify str, int, float, and bool data types.

Types of data in expressions

This skill is associated with data types in Python. Take a look at the following statements:

```
x_position = 2
total = total + 2.5
full_name = 'Rob' + ' ' + 'Miles'
age_valid = age > 18
```

`x_position` is an int, because the value being assigned is an integer

`total` is a float because the valued 2.5 is being added to it (and 2.5 is a floating-point number)

`full_name` is a str because it is made up from strings.

`age_valid` is tricky. It's actually a bool because it is being set to the result of an expression that can be either true or false.

You need to revise these until you can look at an expression and identify the type of the expression. You also need to be able to spot when types are being combined in an incorrect way.

Take a look at the following sections for details of variable storage in Python

Chapter 4: Variables in Python
Chapter 4: Working with text
Chapter 4: Working with numbers
Chapter 5: Boolean data
Chapter 5: Boolean expressions

Perform data and data type operations. Convert from one data type to another type; construct data structures; perform indexing and slicing operations

Data and type operations

Now that we understand that data can be held in variables of a particular type, we need to consider how to convert from one type to another. For example, when reading in a number from a user. The user will type in a string which must be converted into a numeric value:

```
age_string = input('Enter your age: ')
age = int(age_string)
```

The first statement uses the `input` function to read in a string. (note that `age_string` has the type str because that is the type of data that `input` returns). The second statement uses the `int` function to convert this string into an integer value. The `int` function takes a string and returns the int value that the string represents.

```
temp_string = input('Enter the temperature: ')
temp = float(temp_string)
```

These statements repeat the process above, but the `float` function is used to create a floating-point value from the string that the user typed in.

The functions `int`, `float`, `bool` and `str` can be used to convert their arguments into the characteristic types.

Take a look at the following sections for details of variable type conversion:

Chapter 4: Convert strings into floating-point values
Chapter 4: Convert between float and int
Chapter 4: Real numbers and floating-point numbers

Construct data structures

A single variable can hold one value, but frequently you want to store multiple items, for example a collection of sales figures. Python allows you to create a list which can hold multiple values:

```
sales = []
sales.append(30)
sales.append(40)
sales.append(50)
```

The statements above create a list called `sales` which is initially empty. The `append` function is then used to add three elements to the list.

Perform Indexing and Slicing Operations

We can get hold of particular elements of the list by using *indexing*. The list variable is followed by a value in square brackets that gives the **index** of the element to be used.

```
print (sales[1])
```

This will print the content of the *second* element of the list, which would be the value 40. **Lists are indexed starting at zero**

We can also perform "slicing" to create a new list by slicing out a range of values from an existing list:

```
subsales = sales[0:2]
```

This would extract the first two elements (elements 0 and 1) from the list. The value before the colon specifies the index of the start element and the value after the colon specifies the index of the first element **not** to be included in the slice.

Take a look at the following section for an example of how a list can be used to hold data values:

Chapter 8: Lists and tracking sales

Tuples

A tuple can be used to store a collection of items that can be treated as *immutable.* Immutable means that the content of an item cannot be changed. A program can add elements to a list, and change the contents of a list. But a program cannot change the contents of a tuple once it has been created.

```
position = (10,20)
```

The above statement creates a tuple that contains two integers. It could be used to represent the x,y position of an item in 2D space. Note the use of round brackets, which tell Python a tuple is being created.

Take a look at the following sections for an example of how tuples can be used to hold data values:

Chapter 6: The for loop construction
Chapter 8: Tuples

"Determine the sequence of execution based on operator precedence = Assignment; Comparison; Logical; Arithmetic; Identity (is); Containment (in)"

Expression Evaluation

A Python program can contain *expressions* which are evaluated when the program runs. The result of an expression can be assigned to a variable:

```
simple = 2 + 3
```

The above statement contains the expression "2+3", which would be evaluated when the program runs. The statement would apply the + operator between the two operands. The result would be to set the value of the variable `simple` to the integer result 5

Expressions can be more complicated:

```
not_so_simple = 2 + 3 * 4
```

At this point we have to consider whether the expression would set `not_so_simple` to 20 (add 2 and 3 before multiplying) or 14 (multiply 3 by 4 first).

It turns out that multiplication is performed before division, as it would be if we wrote this expression out as a piece of maths. Python enforces operator precedence, which means that multiplication is performed before addition.

Take a look at the following section for details of arithmetic operations:

Chapter 4: Performing calculations

Comparison operations

Programs can perform logical operations that involve the comparison of values. Expressions that contain comparison operators generate logical values as results:

```
age_low = age < 10
```

The statement above will set the variable `age_low` to the logical value `true` if the value in the variable `age` is less than 10.

Take a look at the following section for details of comparison operations:

Chapter 5: Boolean expressions

Logical operations

Logical operators allow programs to combine logical values.

```
can_enter = age_low and paid_fee
```

The `and` operation is applied between two logical expressions. If both expressions are true the result of the and is also true.

There is also an `or` operator.

```
can_enter = age_low and (paid_fee or got_pass)
```

This statement would set `can_enter` to `true` if the age requirement was met and the person had either paid the fee or got a pass).

Take a look at the following section for details of comparison operations:

Chapter 5: Boolean expressions

References and is

Everything in Python is an object. A Python variable is actually a reference to an object. The Python `is` operator can be used to determine if two variables are actually referring to the same object. Consider the existence of a Python class called silly. The following statements create two instances of silly:

```
x = silly()
y = silly()
```

Both silly objects are identical, but they are different objects. One is referred to by the variable `x`, the other by the variable `y`. We can test to see if `x` and `y` refer to the same object by using `is`:

```
the_same = x is y
```

This statement would set the value of `the_same` to `false`, as `x` and `y` refer to different objects. We can set `x` and `y` to refer to the same object if we wish:

```
x=y
```

Now the is test will return `true`, because both `x` and `y` refer to the same object in memory. **Note that the is operation is different from an equals test, because it checks to see if the two references refer to the same object. The is operation doesn't check to see if the objects have the same contents.**

Take a look at the following section for more details on the use of references in programs:

Chapter 9 Contact objects and references

Containment

The `in` operator allows us to check to see if a given Python collection contains a particular value. Given the `sales` list created above, consider the following:

```
got_forty = 40 in sales
```

The variable `got_forty` would be set to `true`, since the `sales` list does contain the value 40. The in operator is most useful when used in conjunction with sets.

Take a look at the following section for more details on the use of sets in Python programs:

Chapter 11 Python sets.

Select the appropriate operator to achieve the intended result: Assignment; Comparison; Logical; Arithmetic; Identity (is); Containment (in).

Here are some scenarios to consider. Think about the operator you need, then look at my answers:

"I want to set the total sales value to zero."

"I want to see if the age is larger than 100."

"I want to see if I am editing the original record or a copy."

"I want to see if a name contains a space."

"I want to make sure that only paid up members with valid registration can enter."

Answers:

"I want to set the total sales value to zero." – use an assignment:

```
total_sales = 0
```

"I want to see if the age is larger than 100." – use a comparison

```
age_large = age > 100
```

"I want to see if I am editing the original record or a copy." – use is

```
same_record = edit_item is original_item
```

"I want to see if a name contains a space." – use in

```
got_space = ' ' in name
```

"I want to make sure that only paid up members with valid registration can enter." – use logical operators

```
can_enter = paid_up and registered
```

Control flow with decisions and loops (25-30%)

To prepare for this topic, you should pay special attention to Chapter 5, which describes the `if` construction, and Chapter 6, which moves on to describe the `while` and `for` loop constructions that are used to implement looping in Python programs.

Skill	Book Element
Construct and analyze code segments that use branching statements.	Chapter 5: Boolean data Chapter 5: The if construction
if; elif; else; nested and compound conditional expressions	Chapter 5: The if construction
Construct and analyze code segments that perform iteration.	Chapter 6: The while construction Chapter 6: The for loop construction
while; for; break; continue; pass; nested loops and loops that include compound conditional expressions.	Chapter 6: The while construction Chapter 6: The for loop construction Chapter 8: Sort using bubble sort

Here's a breakdown of skills which add some more detail and refers to the specific book pages that will deal with each skill:

Construct and analyze code segments that use branching statements

The if construction

The if construction allows the path through a program to vary depending on the values held in variables.

```
if age < 10:
    print("You are too young")
else:
    print("Welcome aboard")
```

The statements above will print "You are too young" if the value in `age` is less than 10. Note the use of indenting to indicate which statements are obeyed if the condition is true.

The statement after the `else` is obeyed if the condition is false, in this case this will be when the value in `age` is greater than 10.

To discover what a given sequence of code will do, work through the code in exactly the same way as the computer would. Keep a note of the contents of variables as the program runs, and use these in the tests.

if; elif; else; nested and compound conditional expressions

Nested conditions

A statements controlled by an if condition can contain if conditions. The code above picks a particular option depending on the value of the command variable. If the command is 1 the command 1 print is performed, and so on. The else clause of each if statement contains a test for the next command, nesting the conditions together.

```
if command==1:
    print("Command 1")
else:
    if command==2:
        print("Command 2")
    else:
        if command==3:
            print("Command 3")
        else:
            print("Invalid command")
```

The elif keyword

A program can use the `elif` keyword to combine the functions of `else` and `if` in a single statement. The `elif` keyword simplifies the above code and removes the need for nesting each successive command.

```
if command==1:
    print("Command 1")
elif command==2:
    print("Command 2")
elif command==3:
    print("Command 3"):
else:
    print("Invalid command")
```

Take a look at the following section for more details on the use of the if construction in Python programs:

Chapter 5: Boolean data
Chapter 5: The if construction

Construct and analyze code segments that perform iteration

Iteration

When you iterate, you "go around again". In program terms this can be because you're working through a large collection of items, or it might just be that your program must repeat indefinitely,

perhaps in the form of a game that updates repeatedly. Iteration is performed in Python by loops of statements.

while; for; break; continue; pass; nested loops and loops that include compound conditional expressions

while

Python provides two iterating constructions, `while` and `for`. The `while` loop will perform statements while a condition is `true`:

```
while true:
    print("Hello")
```

This is a very stupid `while` construction that will print "Hello" forever.

```
while false:
    print("Hello")
```

This is perhaps as stupid, and never prints "Hello" (the test is made before the loop is repeated). The Python compiler is quite happy to compile programs that contain statements that will run forever (or never run).

When the loop ends the statement following the loop is performed:

```
while false:
    print("Hello")
print("This is printed")
```

In the above code the message "This is printed" is always printed as this is not part of the loop.

for

The for construction allows a program to iterate (loop) through a collection of values:

```
for ch in "Hello":
    print(ch)
```

This would print out each letter in the string (each item in the string is one element in the string collection).

range

The `range` function provides a range that a `for` loop can iterate through:

```
for i in range(10):
    print(i)
```

A single parameter specifies a `range` that starts at 10. The parameter gives the first value that is not in the range, so the above program would print values from 0 to 9. The `range` function also accepts start, end and step values:

```
for i in range(1, 21, 2):
    print(i)
```

This program would print out the values 1,3,5,7,9,11,13,15,17 and 19.

break

The code controlled by the iteration can be of any size, and can contain nested loops (iterations) and if conditions. The `break` keyword can be used to break out of a loop when a given condition is detected.

```
for i in range(1, 21, 2):
    if i==9:
        break
    print(i)
```

This code would print out 1,2,3,7 but the loop would then be terminated because the `break` statement controlled by the condition would be performed. Both `for` and `while` loops can contain break statements. A given loop can contain many `break` statements, any one of which would exit the loop. If a `break` is performed in a nested loop it only breaks out of the enclosing loop.

continue

The `continue` statement causes an iteration to restart at the top with the next value. In the case of the `while` loop the condition will be tested before the loop is repeated. In the case of the for loop the loop moves on to the next value, if any.

```
for i in range(1,11):
    if i==9:
        continue
    print(i)
```

The above statement would print out the values 1 to 10, but would not print number 9 as the continue would cause the loop to be restarted by the continue statement.

pass

The `pass` statement can be regarded as a placeholder for code:

```
if age < 10:
    pass
else:
    print("Welcome aboard")
```

In the code above the `pass` statement is performed if the age is less than 10. This doesn't do anything, but it allows this form of `if` construction to be created, as `else` is not allowed to directly follow `if`. The `pass` keyword can be used to represent the body of an empty function.

Take a look at the following section for more details on the use of iteration in Python programs:

Chapter 6: The while construction
Chapter 6: The for loop construction
Chapter 8: Sort using bubble sort

Perform input and output operations (20-25%)

The use of console input and output functions is demonstrated throughout the book, starting with the very first programs described in Chapters 3 and 4. Chapter 8 introduces the use of file storage in Python programs, and Chapter 9 expands on this to show how data structures can be saved into files by the use of the Python pickle library. Chapter 10 contains details of the string formatting facilities available to Python programs.

Skill	Book Element
Construct and analyze code segments that perform file input and output operations. Open; close; read; write; append; check existence; delete; with statement	Chapter 8: Store data in a file Chapter 9: Save contacts in a file using pickle
Construct and analyze code segments that perform console input and output operations. Read input from console; print formatted text; use of command line arguments.	Chapter 3: Get program output using the print function Chapter 4: Read in text using the input function Chapter 10: Python string formatting

Here's a breakdown of skills which add some more detail and refers to the specific book pages that will deal with each skill:

Construct and analyze code segments that perform file input and output operations. Open; close; read; write; append; check existence; delete; with statement

Using files

A file object is created to represent a connection to an existing file. Files can be opened for reading, writing and update.

Take a look at Chapter 8 for details of how to create a file and use it to persist data. The example programs EG8-16 File Output and EG8-18 File Input show how a file is to be used.

Check existence

Python will not stop you from overwriting a file. But you can check for the existence of a file before you write into it, so that program can ask a user to confirm the operation when a file is about to be overwritten. The function to perform the test must be imported from the os.path module.

```
import os.path
if os.path.isfile('test.txt'):
    print('The file exists')
```

You can find out more about this in:

Chapter 8: Write into a file

Deleting a file

The Python `os` module contains a function called `remove` which can be used to remove a file:

```
import os.path
os.remove('test.txt'
```

The statements above delete a file called test.txt. The path to the file is relative to the location of the executing program. To delete files in different folders a complete path must be provided.

The with statement

It is important that files are closed after they have been used, otherwise they may not be written correctly. The `with` statement provides a means by which any object providing a service can be made to "tidy up" once its action is complete. The Python language provides a `with` construction that allows classes to implement a set of behaviors that set up and then complete an operation.

You can find out more about this in:

Chapter 8: Use the with construction to tidy up file access

Construct and analyze code segments that perform console input and output operations. Read input from console; print formatted text; use of command line arguments.

The Python console

The `input` and `print` functions provide input and output for programs. The `input` function is provided with a prompt string that is displayed before the input is read. The `print` function is provided with one or more arguments giving the objects to be printed. Each object is converted into a string before being printed. Note that this is how Python 3.6 `input` and `print` works. In Python 2.7 the `print` and `input` operations are not functions, and so the prompts and items to be printed are not arguments.

A program can use the `format` method to create a formatted string. You can find out more about this in:

Chapter 10: Python string formatting

Command line arguments

When a python program is started from the command line it can be supplied with arguments, for example a program that prints a file can be followed by the name of the file to print:

```
printfile myfile.txt
```

The program can obtain the arguments given by use of the `argv` value in the sys module. This returns a list containing the arguments as issued. The first element of the list is the full path to the program that is running. The second and successive elements are the arguments as given by the user of the program.

```
import sys

with open(sys.argv[1],'r') as input_file:
    print(input_file.read())
```

The above program prints out the file specified by the single argument that is supplied. Note that this should probably check that an argument has been supplied. This can be done by using the `len` function to count the number of items in the argument list:

```
import sys

if len(sys.argv) < 2:
    print("Please enter a filename")
else:
    with open(sys.argv[1],'r') as input_file:
        print(input_file.read())
```

Document and structure code (15-20%)

The importance of well-structured and documented code is highlighted throughout the text. Chapter 3 introduces Python comments, and Chapter 5 contains a discussion highlighting the importance of good code layout. Chapter 7 introduces Python functions in the context of improving program structure and describes how to add documentation to functions to make programs self-documenting.

Skill	Book Element
Document code segments using comments and documentation strings. Use indentation, white space, comments, and documentation strings; generate documentation using pydoc.	Chapter 3: Python comments Chapter 5: Indented text can cause huge problems Chapter 7: Add help information to functions Chapter 12: View program documentation

Construct and analyze code segments that include function definitions: call signatures; default values; return; def; pass.	Chapter 7: What makes a function?

Here's a breakdown of skills which add some more detail and refers to the specific book pages that will deal with each skill:

Document code segments using comments and documentation strings. Use indentation, white space, comments, and documentation strings; generate documentation using pydoc

Indentation

Some programming languages use constructions such as brackets to denote the start and end of blocks of code. Python doesn't do this, instead it uses the indentation of code to indicate to indicate a block. The body of a function will be indented, code controlled by an if construction will be indented, members of a class will be indented.

The precise amount of the indent is left to the programmer, but **it must be consistent within a block**. It is also important that indents do not mix tabs and spaces. This can give rise to code that looks legal, but because some of the statements have been indented using spaces, and others indented using tabs the Python compiler may complain. Fortunately, modern tools can manage indentation for you.

White space

Python programs can contain white space to separate the elements of a program. There is a style guide that sets out the proper use of whitespace in code that you write: https://www.python.org/dev/peps/pep-0008/

Comments

A Python program can contain comments. These are elements of the text that are ignored by the Python compiler. A comment starts with a # (hash) character and extends to the end of the statement.

```
# Version 1.0 by Rob Miles
```

This is a comment that extends over a complete line. A comment can also be started in the middle of a statement:

```
x = 0  # set the cursor to the left margin
```

The comment above is sensible, in that it adds information about the intent of the programmer. Some languages allow programmers to create comments that span several lines. Python does not support this.

Documentation strings

A documentation string can be used to provide extra information about a function, class or module. It is given as the first element in the item. Programs such as pydoc can be used to parse program source and create documentation from these strings. They can also be used by editors to show documentation as programmers work on the code.

```
'These are the stock storage classes'

class StockItem(object):
    ' This class holds information about a stock item'

    def SetPrice(self, new_price):
        'Sets the new price of the stock item'
        pass
```

The above code sample shows each form of documentation string in turn.

Pydoc

Pydoc is supplied as part of a Python installation. It parses source code to produce web pages that display the documentation strings. To see the form of the output that pydoc produces you can take a look here:

Chapter 12: View program documentation

Construct and analyze code segments that include function definitions: call signatures; default values; return; def; pass

Functions

Functions are used to simplify programs. A given behavior can be created once and then used whenever it is needed. Programs can pass arguments into functions and a function can return a single value to a caller. Arguments that are immutable (integers, floating point, bool, string, tuple) are passed by value into the function. Other arguments, for example instances of classes and lists are passed by reference. You can find out more about immutable here:

Chapter 9: Discovering immutable

You can find out more about creating function and using functions here:

Chapter 7: What makes a function?

Perform troubleshooting and error handling (5-10%)

Chapter 3 contains coverage of syntax errors in Python code. In Chapter 4, the description of data processing includes descriptions of runtime errors. In Chapters 6 and 7, the causes and effects of

logic errors are discussed in the context of an application development. Chapters 6 and 10 contain descriptions of how Python programs can raise and manage exceptions, and Chapter 12 contains a description of the use of unit tests in Python program testing.

Skill	Book Element
Analyze, detect, and fix code segments that have errors: Syntax errors; logic errors; runtime errors.	Chapter 3: Broken programs Chapter 4: Typing errors and testing Chapter 5: Equality and floating-point values Chapter 6: When good loops go bad Chapter 7: Investigate programs with the debugger Chapter 12: Program testing
Analyze and construct code segments that handle exceptions: try; except; else; finally; raise.	Chapter 6: Detect invalid number entry using exceptions Chapter 10: Raise an exception to indicate an error

Here's a breakdown of skills which add some more detail and refers to the specific book pages that will deal with each skill:

Analyze, detect, and fix code segments that have errors: Syntax errors; logic errors; runtime errors.

Syntax error

A syntax error is usually easy to fix. Syntax errors are detected by Python when the program contains elements that don't conform to the language design:

```
if age > 10:
   print("You are old")
     print("you can go on the ride")
```

The code above contains a syntax error, in that the second print statement is not indented in a way that is consistent with the first. Other syntax errors include miss-spelling the name of Python elements, using a variable that has not been given a value, missing delimiters off strings, using inconsistent bracketing.

Syntax errors may be picked up before the program runs, but this is not always the case.

Logic error

Logic errors are contained in program code that is syntactically correct, but the actions that the program performs are not right for the context of the operation. Consider a statement intended to reject people who are less than 20 or more than 80 years old:

```
if age<20 and age >80:
    print("reject")
```

This code is completely legal, but contains a logic error in that it will never reject anyone. This is because it is not possible for a variable to contain a value that is both less than 20 and greater than 80.

```
if age<20 or age >80:
    print("reject")
```

This form of error is surprisingly common, and usually produced when a programmer tries to reverse the sense of an action but doesn't do a very good job. Other logic errors include actions like rejecting people with ages on the boundary values (20 or 80) because a program tests for less than when it should test for less than or equals.

Run time error

Run time errors are the result of programming mistakes that don't surface until the program actually runs. They can include things like divide by zero, attempting to access data outside the bounds of a collection, or trying to open a file that is not present. In Python run time errors can be managed by means of exceptions, which allow the programmer to nominate code that runs in the event of a run time error occurring.

Analyze and construct code segments that handle exceptions: try; except; else; finally; raise

Exceptions

Python programs can raise exceptions in the event of a runtime error being detected. A programmer can explicitly catch these and respond to them in a sensible way. Note that when an exception is raised the currently active path through a block is abandoned and execution is transferred to code in the exception handler. It is impossible to return to the statement that raised the exception unless the entire behavior is encapsulated in a loop. You can find out more about exceptions here:

Chapter 7: Create a number input function
Chapter 10: Raising and dealing with exceptions

Finally

The finally statement allows an action to be performed irrespective of whether or not a given block of code generated an exception. They are necessary because code in an exception handler might return from the method that the code is running in, or even generate another exception, which might mean that statements following code protected by an exception may never get to run. You can find out more about finally here:

Chapter 8: Deal with file errors

Perform operations using modules and tools (1-5%)

Many Python modules are used throughout the text, starting with the random and time modules. The functions from the random library are used in Chapter 13 to create random artwork, and functions from the time library are used in a time-tracking application used in Chapter 16.

Skill	Book Element
Perform basic operations using built-in modules: math; datetime; io; sys; os; os.path; random.	Chapter 3: The random library Chapter 3: The time library
Solve complex computing problems by using built-in modules: math; datetime; random.	Chapter 10: Session tracking in Time Tracker Chapter 16: Making art

Here's a breakdown of skills which add some more detail and refers to the specific book pages that will deal with each skill:

Perform basic operations using built-in modules: math; datetime; io; sys; os; os.path; random

Built-in modules

Python provides a number of built-in modules. These are written in Python and provide a huge range of useful functions.

The `math` module provides trigonometric functions (for example `sin`, `cos` `tan`) as well as a huge number of other mathematical functions. The `datetime` object allows programs to work with dates and times, and also to determine the current date and time. It is held in the `time` module, which also holds the `sleep` function. The `io` module provides access to file handling facilities and the `sys` and `os` modules provide access to low level system behaviors. These modules are used in many of the example programs in the text.

os.path

Programs frequently need to manipulate file paths. The os.path module allows a program to extract filenames and directory names from paths, determine if a given path refers to a directory or a file, find the size and date of last access of a file and many other useful functions.

Solve complex computing problems by using built-in modules: math; datetime; random

Functions from these modules are used in many of the sample programs. The random module is used to generate random graphics in:

Chapter 16: Start pygame and draw some lines

Version 1.0 January 2018
Rob Miles