19. Exceptions

19.1. Catching exceptions

Whenever a runtime error occurs, it creates an **exception** object. The program stops running at this point and Python prints out the traceback, which ends with a message describing the exception that occurred.

For example, dividing by zero creates an exception:

```
>>> print(55/0)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

So does accessing a non-existent list item:

```
>>> a = []
>>> print(a[5])
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
IndexError: list index out of range
```

Or trying to make an item assignment on a tuple:

```
>>> tup = ("a", "b", "d", "d")
>>> tup[2] = "c"
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

In each case, the error message on the last line has two parts: the type of error before the colon, and specifics about the error after the colon.

Sometimes we want to execute an operation that might cause an exception, but we don't want the program to stop. We can **handle the exception** using the `try` statement to "wrap" a region of code.

For example, we might prompt the user for the name of a file and then try to open it. If the file doesn't exist, we don't want the program to crash; we want to handle the exception:

```
1  filename = input("Enter a file name: ")
2  try:
3      f = open(filename, "r")
4  except:
5      print("There is no file named", filename)
```

The `try` statement has three separate clauses, or parts, introduced by the keywords `try` ... `except` ... `finally`. Either the `except` or the `finally` clauses can be omitted, so the above code considers the most common version of the `try` statement first.

The `try` statement executes and monitors the statements in the first block. If no exceptions occur, it

skips the block under the `except` clause. If any exception occurs, it executes the statements in the `except` clause and then continues.

We could encapsulate this capability in a function: `exists` which takes a filename and returns true if the file exists, false if it doesn't:

```
1  def exists(filename):
2      try:
3          f = open(filename)
4          f.close()
5          return True
6      except:
7          return False
```

A template to test if a file exists, without using exceptions

The function we've just shown is not one we'd recommend. It opens and closes the file, which is semantically different from asking "does it exist?". How? Firstly, it might update some timestamps on the file. Secondly, it might tell us that there is no such file if some other program already happens to have the file open, or if our permission settings don't allow us to open the file.

Python provides a module called `os.path` within the `os` module. It provides a number of useful functions to work with paths, files and directories, so you should check out the help.

```
1  import os
2
3  # This is the preferred way to check if a file exists.
4  if os.path.isfile("c:/temp/testdata.txt"):
5      ...
```

We can use multiple `except` clauses to handle different kinds of exceptions (see the Errors and Exceptions lesson from Python creator Guido van Rossum's Python Tutorial for a more complete discussion of exceptions). So the program could do one thing if the file does not exist, but do something else if the file was in use by another program.

19.2. Raising our own exceptions

Can our program deliberately cause its own exceptions? If our program detects an error condition, we can **raise** an exception. Here is an example that gets input from the user and checks that the number is non-negative:

```
1  def get_age():
2      age = int(input("Please enter your age: "))
3      if age < 0:
4          # Create a new instance of an exception
5          my_error = ValueError("{0} is not a valid age".format(age))
6          raise my_error
7      return age
```

Line 5 creates an exception object, in this case, a `ValueError` object, which encapsulates specific information about the error. Assume that in this case function `A` called `B` which called `C` which called `D` which called `get_age`. The `raise` statement on line 6 carries this object out as a kind of "return value",

and immediately exits from `get_age()` to its caller `D`. Then `D` again exits to its caller `C`, and `C` exits to `B` and so on, each returning the exception object to their caller, until it encounters a `try ... except` that can handle the exception. We call this "unwinding the call stack".

`ValueError` is one of the built-in exception types which most closely matches the kind of error we want to raise. The complete listing of built-in exceptions can be found at the Built-in Exceptions section of the Python Library Reference , again by Python's creator, Guido van Rossum.

If the function that called `get_age` (or its caller, or their caller, ...) handles the error, then the program can carry on running; otherwise, Python prints the traceback and exits:

```
>>> get_age()
Please enter your age: 42
42
>>> get_age()
Please enter your age: -2
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "learn_exceptions.py", line 4, in get_age
    raise ValueError("{0} is not a valid age".format(age))
ValueError: -2 is not a valid age
```

The error message includes the exception type and the additional information that was provided when the exception object was first created.

It is often the case that lines 5 and 6 (creating the exception object, then raising the exception) are combined into a single statement, but there are really two different and independent things happening, so perhaps it makes sense to keep the two steps separate when we first learn to work with exceptions. Here we show it all in a single statement:

```
1  raise ValueError("{0} is not a valid age".format(age))
```

19.3. Revisiting an earlier example

Using exception handling, we can now modify our `recursion_depth` example from the previous chapter so that it stops when it reaches the maximum recursion depth allowed:

```
1  def recursion_depth(number):
2      print("Recursion depth number", number)
3      try:
4          recursion_depth(number + 1)
5      except:
6          print("I cannot go any deeper into this wormhole.")
7
8  recursion_depth(0)
```

Run this version and observe the results.

19.4. The `finally` clause of the `try` statement

A common programming pattern is to grab a resource of some kind — e.g. we create a window for turtles to draw on, or we dial up a connection to our internet service provider, or we may open a file for writing. Then we perform some computation which may raise an exception, or may work without any

problems.

Whatever happens, we want to "clean up" the resources we grabbed — e.g. close the window, disconnect our dial-up connection, or close the file. The `finally` clause of the `try` statement is the way to do just this. Consider this (somewhat contrived) example:

```
1  import turtle
2  import time
3
4  def show_poly():
5      try:
6          win = turtle.Screen()   # Grab/create a resource, e.g. a window
7          tess = turtle.Turtle()
8
9          # This dialog could be cancelled,
10         #   or the conversion to int might fail, or n might be zero.
11         n = int(input("How many sides do you want in your polygon?"))
12         angle = 360 / n
13         for i in range(n):      # Draw the polygon
14             tess.forward(10)
15             tess.left(angle)
16         time.sleep(3)           # Make program wait a few seconds
17     finally:
18         win.bye()               # Close the turtle's window
19
20 show_poly()
21 show_poly()
22 show_poly()
```

In lines 20–22, `show_poly` is called three times. Each one creates a new window for its turtle, and draws a polygon with the number of sides input by the user. But what if the user enters a string that cannot be converted to an `int`? What if they close the dialog? We'll get an exception, *but even though we've had an exception, we still want to close the turtle's window*. Lines 17–18 does this for us. Whether we complete the statements in the `try` clause successfully or not, the `finally` block will always be executed.

Notice that the exception is still unhandled — only an `except` clause can handle an exception, so our program will still crash. But at least its turtle window will be closed before it crashes!

19.5. Glossary

exception

An error that occurs at runtime.

handle an exception

To prevent an exception from causing our program to crash, by wrapping the block of code in a `try` ... `except` construct.

raise

To create a deliberate exception by using the `raise` statement.

19.6. Exercises

1. Write a function named `readposint` that uses the `input` dialog to prompt the user for a positive integer and then checks the input to confirm that it meets the requirements. It should be able to handle inputs that cannot be converted to `int`, as well as negative `int`s, and edge cases (e.g. when

the user closes the dialog, or does not enter anything at all.)