DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

CSC 243 NOTES
Fall 2020: Amber Settle

Monday, October 5, 2020

Announcements

- Questions about the third quiz or the second assignment?
- The third assignment is due Thursday, October 8th at 9 pm – questions?

Objects

In Python, every value, whether a simple integer value (like 3) or a more complex value (such as the list ['hello', 4, 5]) is stored in memory as an **object**.

Associated with every Python object are:

1. A **type** that indicates what kind of values the object can hold – this is important because the type determines what kind of operations are valid for the object
2. A **value**, i.e. the contents or the data

To illustrate the difference, consider the following examples:

```
>>> type(3)
<class 'int'>
>>> type(3.0)
<class 'float'>
>>> type('hello')
<class 'str'>
>>> type([])
<class 'list'>

>>> 3
3
>>> 3.0
3.0
>>> 'hello'
'hello'
>>> []
[]
```

There are two types of comparisons you can do between objects:

1. **Object value comparisons**: Compare the values of two objects of the same type
2. **Object identity comparisons**: Compare the objects themselves

To see how this works, consider some examples.

Object value comparisons

Comparison operators are used to determine the equality between two objects of the same type. Some examples are found below:

```
>>> 3 == 3
True
>>> 3 == 4
False
>>> [2, 3, 4] == [2, 3, 4]
True
>>> [2, 3, 4] == [3]
False
>>> 3 <= 4 == 4 < 8 != 9
True
```

Object identity comparisons

In Python you can compare objects themselves using the **is** keyword.

Two objects are equal if they are the same object, i.e. they have the same id. Some examples are given below:

```
>>> a = [3, 4]
>>> b = a
>>> a == b
True
>>> a is b
True
>>> b is a
True
>>> c = [3, 4]
>>> a == c
True
>>> a is c
False
```

Sometimes the Python interpreter does unexpected things:

```
>>> a = 2.0
>>> b = 2.0
>>> a is b
False
>>> a = 2
>>> b = 2
>>> a is b
True
```

The Python virtual machine decided, on its own, to save some memory and re-use integer objects with value 2. This is actually fine because numbers are immutable.

Loop pattern: Infinite loop

The while loop can be used to create an “infinite” loop, i.e. a loop that runs forever.

```
while True:
      <body>
```

This loop can only be broken by typing (simultaneously) Ctrl-C on the keyboard.

Infinite loops are useful when the program provides a service indefinitely.

A web server, i.e. a program that serves web pages, is an example of a program that provides a service.

Practice problem: Write a function **hello**() that repeatedly requests that the user input their name and then greets the user using that name. If the user enters without typing anything, the function stops.

The following is an example of how the function would be used:

```
>>> hello()
What's your name? Amber
Hello Amber!
What's your name? Marco
Hello Marco!
What's your name?
```

See the solution in **solutions.py**.

Loop pattern: Interactive loop

A while loop is useful when a program needs to **repeatedly request input from the user**, until the user enters a flag which is a value indicating the end of the input.

Example: Write a function **interact**() that repeatedly requests positive integers from the user and appends them to a list. The user will indicate the end of the input with a non-positive value (<= 0), at which point the function should return the list of all positive integers entered by the user.

The following is an example of how the function would be used:

```
>>> interact()
Enter value: 3
Enter value: 1
Enter value: 8
Enter value: 0
[3, 1, 8]
>>> interact()
Enter value: -1
[]
```

The solution uses the accumulator loop pattern in addition to the interactive loop pattern. See the answer in the **solutions.py** file.

Exceptions

There are two basic types of errors that can occur when running a Python program.

Syntax errors are errors that are due to the incorrect format of a Python statement.

These errors occur while the statement or program is being parsed and before it is being executed.

A Python Virtual Machine tool called a parser discovers these errors.

```
>>> (3+4]
SyntaxError: invalid syntax
>>> if x == 5
SyntaxError: invalid syntax
>>> print 'hello'
SyntaxError: invalid syntax
>>> lst = [4;5;6]
SyntaxError: invalid syntax
```

Built-in exceptions occur during the execution of the statement or program.

They do not occur because of a malformed Python statement or program but rather because the program execution gets into an erroneous state.

For example, the following is an error caused by **division by 0**:

```
>>> 4/0
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    4/0
ZeroDivisionError: int division or modulo by
zero
```

The following is an error caused by an **invalid list index**:

```
>>> lst = [14, 15, 16]
>>> lst[3]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    lst[3]
IndexError: list index out of range
```

This is an error cause by an **unassigned variable name**:

```
>>> x+5
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    x+5
```

```
NameError: name 'x' is not defined
```

The following is an error caused by an **illegal value**:

```
>>> int('4.5')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    int('4.5')
ValueError: invalid literal for int() with base
10: '4.5'
```

In each case an error occurs because the statement execution got into an invalid state. When this happens, we say that the Python interpreter **raises an exception**.

What this means is that an object gets created and this object will contain all the information relevant to the error.

For example, it will contain the error message that indicates what happened and the program (module) line number at which the error occurred.

When an error occurs the default is for the statement or program to "crash" and for a message containing all this information to be printed.

The object created when an error occurs is called an **exception**.

Every exception has a type that is related to the type of error.

For example, `ZeroDivisionError`, `IndexError`, `NameError`, `TypeError`, `ValueError`, and many others.

Catching and handling exceptions

Some programs should not terminate when an exception occurs: server programs, shell programs, etc.

The reason why error objects are called exceptions is because when they get created:

- The normal execution flow of the program (as described by, say, the program's flowchart) stops, and
- The execution switches to the so-called exceptional control flow

The default exceptional control flow is to stop the program and print the error message contained in the exception object.

We can change this default exceptional control flow using the try/except pair of clauses.

Compare this:

```
>>> strAge = input('Enter your age: ')
Enter your age: 22
>>> age = int(strAge)
```

With:

```
>>> strAge = input('Enter your age: ')
Enter your age: fifteen
>>> age = int(strAge)
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    age = int(strAge)
ValueError: invalid literal for int() with base
10: 'fifteen'
```

With:

```
>>> strAge = input('Enter your age: ')
Enter your age: fifteen
>>> try:
      age = int(strAge)
except:
      print('Enter your age using digits 0-9!')


Enter your age using digits 0-9!
```

With:

```
>>> strAge = input('Enter your age: ')
Enter your age: fifteen
>>> try:
      age = int(strAge)
except ValueError:
      print('Enter your age using digits 0-9!')


Enter your age using digits 0-9!
```

When an exception is encountered while executing the body of a try statement, the Python interpreter will jump to the exception handler, i.e. the body of the except statement associated with the try statement.

The **try statement works as follows**:

- First, the try clause (the statements(s) between the try and except keywords) is executed.
- If no exception occurs, the except clause is skipped and the execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception

named after the except keyword, the except clause is executed and then execution continues after the try statement.

- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an **unhandled exception** and execution stops with a message as shown above.

Multiple except clauses

A try statement may have more than one except clause, to specify handlers for different exceptions.

An except clause may also name multiple exceptions as a parenthesized tuple:

```
except (RuntimeError, TypeError, NameError):
      pass
```

When an exception occurs, at most one handler will be executed, namely the one that deals with the particular exception type.

The last except clause in a sequence of except clauses may omit the exception name(s), to serve as a wildcard. It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

```
try:
      f = open('mfile2.txt', 'r')
      s = f.readline()
      i = eval(s)
except IOError as err:
      print("I/O error {}".format(err))
except ValueError:
      print("Could not convert data to an
      integer.")
except Exception as err:
      print("Unexpected error:{}".format(err))
```

This code can be found in the file **exceptions.py**.

Finally, if an exception is raised in one of the exception-handling clauses, the exception will be passed on:

```
try:
      x = int(input('Enter a number: '))
except ValueError:
      print('That was not valid.')
      x = int(input('Enter a number: '))
except:
      print('A generic error')
```

Again see the file **exceptions.py** for this code.

How can we change the example above to force the entry of an integer?

Make modifications in the **exceptions.py** file.

Examples of handling exceptions

Considering some specific problems will help us to understand exceptions better.

Practice problems:

1. Write a version of the **printNth**() function from the first assignment that uses exception handling to ensure that the index the user enters is valid for every string.

   Start with the solution in the **exceptions.py** file.
   Do not use an if statement!

   The following shows some sample interactions:

```
>>> printNth(['one', 'two', 'three'])
Enter a whole number: 2
three
>>> printNth(['one', 'two', 'three'])
Enter a whole number: 3
>>> printNth(['one', 'two', 'three'])
Enter a whole number: -3
one
>>> printNth(['one', 'two', 'three'])
Enter a whole number: -4
>>>
```

2. Write a function **safeOpen**() that inputs the name of a file from the user. It attempts to open the file, and if it succeeds returns the opened file. If it fails to open the file, it again asks the user for the name of a file and tries to open and return it. It continues this process until the user provides a file that can be correctly opened and returned.

   The following shows some sample interactions, assuming that the files **none1.txt** and **none2.txt** do not exist in the current directory, but the file **sample.txt** does:

```
>>> inf = safeOpen()
Enter a file name: none1.txt
none1.txt could not be opened.
Please try again.
Enter a file name: none2.txt
none2.txt could not be opened.
Please try again.
Enter a file name: mfile2.txt
>>> print(inf.read())
17.2
>>> inf.close()
```

```
>>>
```

See the solutions in **exceptSols.py**.

The random module

Random numbers are useful in a variety of contexts:

- Simulations for science, engineering, and finance
- Computer security protocols
- Computer games (e.g. poker, blackjack, digital games)

Unfortunately the machines we use are deterministic, which means (among other things) they have no access to genuine randomness.

Instead we have to make do with **pseudo random numbers**, i.e. "almost" random numbers.

> A pseudo random number generator produces a sequence of numbers that "look" random and are good enough for most applications that need random numbers.

To use a pseudo random number generator in Python, we need to import the random module:

```
>>> import random
```

Functions in the random module

There are a number of functions in the random module that are helpful.

Choosing a random integer

The function **randrange**() takes a pair of integers and returns some number in the range [a, b), that is including a but not including b.

```
>>> random.randrange(1,7)
3
>>> random.randrange(1,7)
4
>>> random.randrange(1,7)
6
>>> random.randrange(1,7)
1
>>> random.randrange(1,7)
5
```

Choosing a random decimal

Sometimes we need a decimal random number rather than a decimal integer.

In that case we need to use the **uniform**() function which takes two numbers a and b and returns a float number x such that $a \leq x \leq b$ (assuming that $a \leq b$), with each float in the range equally likely.

The following obtains several random numbers between 0 and 1:

```
>>> random.uniform(0,1)
0.7378384739678002
>>> random.uniform(0,1)
0.09911455205729514
>>> random.uniform(0,1)
0.18295866524385507
>>> random.uniform(0,1)
0.3319868023085931
```

Shuffling, choosing and sampling

The function **shuffle**() shuffles, or permutes, the objects in a sequence something like a deck of cards is shuffled before playing a card game. Each permutation of the object is equally likely.

```
>>> lst = [1, 2, 3, 4, 5]
>>> random.shuffle(lst)
>>> lst
[1, 2, 4, 5, 3]
>>> random.shuffle(lst)
>>> lst
[3, 5, 2, 1, 4]
>>> words = ['cat', 'bat', 'at', 'fat']
>>> random.shuffle(words)
>>> words
['at', 'cat', 'bat', 'fat']
```

The function **choice**() allows us to choose an item from a container uniformly at random:

```
>>> random.choice(lst)
5
>>> random.choice(lst)
4
>>> random.choice(lst)
4
>>> random.choice(lst)
1
>>> random.choice(words)
'fat'
>>> random.choice(words)
'cat'
>>> random.choice(words)
'cat'
>>> random.choice(words)
'fat'
```

We use the **sample**() function if instead of choosing a single item from a sequence we wanted to choose a sample of size k, with every sample equally likely. The function takes the container and the number k as parameters:

```
>>> random.sample(lst, 2)
[1, 2]
>>> random.sample(lst, 2)
[4, 2]
>>> random.sample(lst, 2)
[3, 4]
>>> random.sample(words, 3)
['at', 'cat', 'bat']
>>> random.sample(words, 3)
['fat', 'at', 'cat']
>>> random.sample(words, 2)
['fat', 'bat']
```

Practice problem: Implement a function **guess**() that takes an integer n as a parameter and implements a simple, interactive guessing game.

The function should start by choosing a random number in the range from 1 up to and including n.

The function should then repeatedly ask the user to enter to guess the chosen number. When the user guesses correctly, the function should print a 'You got it' message and terminate. Each time the user guesses incorrectly, the function should indicate whether the guess was too high or too low.

```
>>> guess(100)
Enter your guess: fifty
That was not a valid number.
Enter your guess: 5.5
That was not a valid number.
Enter your guess: -1
That was not a valid number.
Enter your guess: 101
That was not a valid number.
Enter your guess: 50
Too low
Enter your guess: 75
Too low
Enter your guess: 87
Too low
Enter your guess: 95
Too high
Enter your guess: 90
Too high
Enter your guess: 88
Too low
Enter your guess: 89
You got it
```

See the solution in **randomEx.py**.