Lesson 7 - Thomas Deneuville

November 11, 2017

1 `sorted` function and Exceptions

Python has a useful tool to sort elements in an iterable - lists for example. Use the `sorted` function.

How `sorted` works is that you specify an iterable and it sorts the elements in the iterable. It returns a new iterable that is the sorted version of the input.

```
In [3]: A = [-1, 3, 5, 100, 8, -100, 9, 6]
        B = sorted(A)

        print(A)
        print(B)
```

```
[-1, 3, 5, 100, 8, -100, 9, 6]
[-100, -1, 3, 5, 6, 8, 9, 100]
```

If we provide more complicated iterables, like a list of tuples for example, what Python will do is that it will sort based on the first index of the iterable inside the list of iterables.

```
In [5]: C = [(-1, -1), (6, 100), (-10, 4)]
        D = sorted(C)
        print(C)
        print(D)
```

```
[(-1, -1), (6, 100), (-10, 4)]
[(-10, 4), (-1, -1), (6, 100)]
```

```
In [7]: E = [(-1, -1, 100), (6, 100, 89898), (-100, 4, 19191), (99, 89, 76)]
        F = sorted(E)
        print(E)
        print(F)
```

```
[(-1, -1, 100), (6, 100, 89898), (-100, 4, 19191), (99, 89, 76)]
[(-100, 4, 19191), (-1, -1, 100), (6, 100, 89898), (99, 89, 76)]
```

We can optionally specify which element inside each nested iterable we would choose to sort our list. In this case, you specify the `key` parameter which is an anonymous function or function that determines which element to serve as the way of dictating the sort order.

For example, if we wanted to use the second element as the way of sorting, do:

```
In [8]: G = sorted(E, key=lambda x: x[1]) # x is expected to be an element inside the list
                                          # In this case, it is a tuple of three elements
                                          # Choose the second element to dictate our order
        print(E)
        print(G)
```

```
[(-1, -1, 100), (6, 100, 89898), (-100, 4, 19191), (99, 89, 76)]
[(-1, -1, 100), (-100, 4, 19191), (99, 89, 76), (6, 100, 89898)]
```

This is very powerful especially for mixed types. Let's say we had a list of tuples where the first element is a name and the second element is a grade. Let's say we wanted to sort the grades in ascending order and show the rank of each student. We could figure out the largest or smallest grade and so on.

```
In [11]: X = [("Ray", 89), ("Thomas", 100), ("June", 70), ("May", 34), ("Jane", 76)]

         # Print out the order as alphabetical by name
         Xs = sorted(X, key=lambda x:x[0])
         print(Xs)

         # Print out the order based on the grade
         Xs2 = sorted(X, key=lambda x:x[1])
         print(Xs2)
```

```
[('Jane', 76), ('June', 70), ('May', 34), ('Ray', 89), ('Thomas', 100)]
[('May', 34), ('June', 70), ('Jane', 76), ('Ray', 89), ('Thomas', 100)]
```

We can also sort in **reverse** order. You simply specify the `reverse` parameter to `True`.

```
In [12]: Xs3 = sorted(X, key=lambda x: x[1], reverse=True)
         print(Xs3)
```

```
[('Thomas', 100), ('Ray', 89), ('Jane', 76), ('June', 70), ('May', 34)]
```

Unexpected things may occur if you try to sort nested iterables where the same position over different elements are different types.

```
In [14]: X = [("Ray", "0"), ("Thomas", 100), ("June", (1, 2, 3)), ("May", [0]), ("Jane", -1)]
         print(sorted(X, key=lambda x:x[1]))

         # Not sure how to order a string and integer.
         # That's why you get a TypeError.
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-14-c393ea8df6ad> in <module>()
  1 X = [("Ray", "0"), ("Thomas", 100), ("June", (1, 2, 3)), ("May", [0]), ("Jane", -1)]
----> 2 print(sorted(X, key=lambda x:x[1]))
  3
  4 # Not sure how to order a string and integer.
  5 # That's why you get a TypeError.


TypeError: unorderable types: int() < str()
```

2 Exceptions

An exception is an error tha thappens during the execution of your program. When that error occurs, Python generates an **exception** and that usually halts the execution of your program.

When we do **exception handling** this means that we can handle the exception in a nicer way and prevent our program from halting.

3 Why use exceptions?

Exceptions are a convenient way for handling errors and special conditions in a program. When you think you have code that can produce a potential error, it's good practice to use exceptions.

```
In [15]: A = 1 / 0
```

```
---------------------------------------------------------------------------

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-15-3f91814b6b8a> in <module>()
----> 1 A = 1 / 0


ZeroDivisionError: division by zero
```

When there is potential problematic code that may create an exception or an error, put it in a `try` statement. Any exceptions that you expect to happen, you would use the `except` keyword.

```
In [19]: try:
             A = 1/0
         except ZeroDivisionError:
             print("You cannot divide by 0!")
```

```
You cannot divide by 0!
```

The `except` keyword - you place the error that you are looking for beside the keyword. If that exception happens, then the code below the `except` keyword is what executes instead.

```
In [24]: # Any code that is potentially after an exception in the try statement, it does not run
         try:
             A = int(input('Enter a number: '))
             print(A)
         except ValueError:
             print('This is not a number!')
```

```
Enter a number: -100
-100
```

Let's make this a bit more complicated. Let's first check if it's an integer, and let's see if we can find the reciprical.

```
In [27]: try:
             A = input('Enter a number: ')
             # Try and convert the number into an integer
             A = int(A) # ValueError may occur
             # Let's try to find the reciprical
             B = 1 / A
             print(A, B)
         except ValueError:
             print('A is not an integer!')
         except ZeroDivisionError:
             print('A cannot be 0!')
```

```
Enter a number: -100
-100 -0.01
```

Some people like to just use the keyword `Exception` beside the `except` keyword. That way, no matter exception happens, we will be able to catch it.

Let's have a more complicated example where we will keep asking for input until it is legitimate.

```
In [28]: while True:
             try:
                 # Get the number from the user as a string
                 A = input('Enter a number from 1 to 10: ')
                 # First convert the number
                 A = int(A)
                 if 1 <= A <= 10:
```

```
                break
            else:
                print("The number is not between 1 and 10.  Try again.")
        except ValueError:
            print("You did not enter in a valid number!  Try again")
```

```
Enter a number from 1 to 10: 0
The number is not between 1 and 10.  Try again.
Enter a number from 1 to 10: -1
The number is not between 1 and 10.  Try again.
Enter a number from 1 to 10: asdfasdf
You did not enter in a valid number!  Try again
Enter a number from 1 to 10: 8
```