Data types and data structures

A *data type* (also called *abstract data type* or *ADT*) defines the operations and the behavior supported by an object. A data type is a *concept*, similar to mathematical concepts such as "function," "set," or "sequence."

It is important not to confuse the terms *data type* and *data structure*. A *data structure* is an *implementation* of a data type: An object that provides all the operations defined by the data type, with the correct behavior.

Quite often, there are different possible implementations for the same data type. For instance, stacks can be implemented with arrays or with linked lists, sets can be implemented with search trees or with hash tables.

This course is called *Data Structures*, but a better name would be *Data Types and Data Structures*. We will meet several important data types, in particular *stack*, *queue*, *set*, *priority queue*, and *map*. We will also discuss various data structure techniques that can be used to implement these data types. In particular, we will meet linked lists, trees, balanced trees, and hash tables.

When considering a problem, it is a good idea to think about the data types that will be used to solve the problem. To define an abstract data type, you need to

- specify the possible values that the data type can have;
- define the operations that can be performed on the data type.

You can then solve the problem at hand by making use of your new data type—ignoring its implementation details.

Being careful about defining the allowed operations stops clients of the data type from accessing the implementation, and guards against logical errors.

Perhaps most important is the ability to change the implementation of a data type later, when you discover that your original implementation is too slow, or has some other disadvantage. As long as the new implementation provides the same operations and behavior, the client code can remain unchanged.

Finally, abstract data types provide a clean and systematic way of dividing large programs into smaller, manageable modules, that can perhaps be implemented by different programmers or teams.

An example: a day calculator

Let's go through one example. We want to implement a day calculator, that can tell you several things:

- For a given date, what day of the week is it?
- For two given dates, how many days lie between?
- For a given date and a given number of days, what date is that many days before or after the given date?

Clearly, to write this program, it will be useful to have objects that represent *dates*. We define an abstract data type `Date` as follows:

- `Date(yr, m, d)` create a new date object.
- `day()` return the day.
- `month()` return the month.
- `year()` return the year.
- `dayOfWeek()` return the day of the week as a number 0...6 (0 is Monday).
- `numDays(otherDate)` return the number of days between the two dates.
- `advanceBy(n)` return date n days further (or earlier, if n is negative).

In addition, we want to be able to compare dates using `==` and `<`.

Once we have defined this ADT, we can actually write client code (remember, this means code that *uses* the `Date` data type). For instance:

```
>>> a = Date(1996, 9, 3)
>>> b = Date(2015, 9, 8)
>>> a.numDays(b)
6944
>>> print(a.advanceBy(7000))
2015/11/03
```

First implementation

Let's now implement the Date data type. An obvious implementation stores year, month, and day of a data in fields of the object. The constructor and the first three methods would look like this:

```
class Date():
  def __init__(self, year, month, day):
    self._year = year
    self._month = month
    self._day = day

  def year(self):
    return self._year

  def month(self):
    return self._month

  def day(self):
    return self._day
```

To implement dayOfWeek and numDays, we need to be able to convert dates into a day number that simply counts days since some fixed date in the past. Using a private method _toJulianDay that converts a date to a Julian day number, we can implement them like this:

```
  def dayOfWeek(self):
    jday = self._toJulianDay()
    return jday % 7

  def numDays(self, otherDate):
    return otherDate._toJulianDay() - self._toJulianDay()
```

To implement advanceBy, we need another private function that converts a Julian day number back to a Date object:

```
  def advanceBy(self, days):
    jday = self._toJulianDay() + days
    y, m, d = _jdayToYMD(jday)
    return Date(y, m, d)
```

Client program

With our first implementation (in the example code as date1.py), we can now write and test the day calculator. You can find my implementation in the example code as days1.py.

Alternative implementation

The implementation in date1.py works fine, but it's not always the best solution. Consider a situation where you need to store millions of Date objects. In this case it would be best to make the Date object as small as possible. Instead of storing three integers for year, month, and day, we could just store a single integer—the Julian day number.

The file date2.py implements this solution:

```
class Date():
  def __init__(self, year, month, day):
    self._jday = _toJulianDay(year, month, day)

  def _toYMD(self):
    return _jdayToYMD(self._jday)
  def year(self):
    return self._toYMD()[0]

  def month(self):
    return self._toYMD()[1]

  def day(self):
    return self._toYMD()[2]
```

In this implementation, the constructor needs to convert from year, month, and day to Julian day number, and each of the accessor methods needs to convert back. On the other hand, `dayOfWeek` and `numDays` become very easy now:

```
  def dayOfWeek(self):
    return self._jday % 7

  def numDays(self, otherDate):
    return otherDate._jday - self._jday
```

We have now two working implementations of the `Date` data type. Both can be used equally well with our day calculator, and we can switch between them by changing just one `import` line in the day calculator program.

This is the power of abstract data types: Since the client code only knows about the data type definition, it works equally well with any implementation of the data type.

Comparisons

We have not yet added the comparison operations to our implementations. It's slightly easier doing this for the second implementation, since we can just compare the Julian day numbers. It suffices to define a few magic methods:

```
  def __eq__(self, rhs):
    return self._jday == rhs._jday

  def __lt__(self, rhs):
    return self._jday < rhs._jday

  def __le__(self, rhs):
    return self._jday <= rhs._jday
```

The `__eq__` method implements equality, and makes the `==` and `!=` operators work correctly. The `__lt__` method implements the "less than" operator `<`, while the `__le__` method implements the "less than or equal" operator `<=`. Python is smart enough to also use these methods for the `>` and `>=` operators.

Exceptions

Neither of our `Date` implementations handled invalid dates in the constructor:

```
>>> from date2 import Date
>>> d = Date(2017, 8, 32)
```

We would like to print an error message when the user enters such a date, but our `Date` data type accepts it silently.

It's actually quite easy to detect invalid dates: we can simply convert year, month, and day to the Julian day number, then convert it back and check if we get the same values.

The problem is that the constructor always returns a `Date` object—so how can we report that the arguments are invalid? The solution is to *raise an exception*:

```
  def __init__(self, year, month, day):
    jday = _toJulianDay(year, month, day)
    y, m, d = _jdayToYMD(jday)
    if y != year or m != month or d != day:
      raise ValueError("Invalid Gregorian date")
    self._jday = jday
```

When using this definition, invalid dates no longer work:

```
>>> from date4 import Date
>>> d = Date(2017, 8, 32)
ValueError: Invalid Gregorian date
```

But now we also need to handle the exception in our client code, otherwise the program will simply crash when the user enters an invalid date. This is done using the `try` and `except` keywords (see `days2.py`):

```
  while True:
    s = input("> ")
    f = s.split()
    try:
      if len(f) == 0:
        return
      elif len(f) == 1:
        show_weekday(f[0])
      elif len(f) == 2:
        show_difference(f[0], f[1])
      elif len(f) == 3:
        show_advance(f[0], f[1], f[2])
      else:
        print("Incorrect command")
    except ValueError as e:
      print(e)
```

If an exception happens *anywhere* inside the `try`-block, including in any function called from the `try`-block (such as `show_weekday` and then the `Date` constructor), then execution is aborted, and continues after the `except`-clause that matches the exception. Since the `Date`-constructor raises a `ValueError` exception, execution continues by printing the error message. The loop then continues to ask the user for the next day calculation.

More about exceptions

In the previous section, we saw how to use exceptions in our day calculator. Let's now discuss them in some more detail.

If you are like me, you have seen many times how your programs terminate with an error or exception message. Here are some examples of *exception messages*:

```
>>> a = 3
>>> a // 0
ZeroDivisionError: integer division or modulo by zero
>>> s = "abc"
>>> int(s)
ValueError: invalid literal for int() with base 10: 'abc'
>>> f = open("test.txt", "r")
FileNotFoundError: [Errno 2] No such file or directory: 'test.txt'
>>> data = [ None ] * 10000000000
MemoryError
```

Some exceptions, such as `MemoryError`, indicate a serious failure, where continuing the program makes no sense.

Other *exceptions*, however, merely indicate an *unexpected* or *abnormal* condition in a program. For instance, a mistake in the input data of a program could cause an exception. Such mistakes can be handled: We say that the exception is *handled* or *caught.*

For instance, a `ValueError` might indicate that the user entered an incorrect number, and the correct response would be to print an error message and ask for new input.

A `FileNotFoundError` means that the file we tried to open does not exist. Depending on the situation, the correct response could be to try a different file name, to ask the user for a different file name, or simply to skip reading the file.

Catching exceptions The following code asks the user for a number. The standard function `input` returns a string, so we have to convert that to an integer using the `int` function. If the string is not a number, such as "abc" or "123ab", then the `int` function *raises* an exception. We can *catch* the exception by enclosing the critical part in a `try`-block, and adding an `except`-clause to handle the exceptions we are interested in:

```
s = input("Enter an integer> ")

try:
  x = int(s)
  print("You said: %g" % x)
except ValueError:
  print("'%s' is not a number" % s)
```

If the `try`-block executes normally, then the `except`-clauses are skipped. But if somewhere inside the `try`-block (including in any method called, directly or indirectly) an exception is thrown, then execution of the `try`-block stops immediately, and continues in the first `except`-clause that *matches* the exception. Here, "matches" means that the exception is the same type as the exception type listed in the clause.

The code within an `except`-clause is called an *exception handler.*

In our example above, if the string `s` does not represent an integer (for instance, if it is "abc"), then `int(s)` throws the exception `ValueError`. The `try`-block is terminated (and in particular, no value is assigned to `x`), and execution continues in the `except`-clause for `ValueError`. Here are some example runs:

```
$ python3 catch1.py
Enter an integer> 123a
'123a' is not a number
$ python3 catch1.py
Enter an integer> -234
You said: -234
```

Exceptions versus error codes Old programming languages like C do not have exceptions, and so all errors or unusual conditions need to be handled by *error codes.* In C++, error codes are also still widely used, for instance for compatibility with C.

A simple and elegant function call like `int(s)` is impossible without exceptions. We would have to return *two results*: one *Boolean* value to indicate whether the conversion was successful, and the integer value itself.

So exceptions allow us to concentrate on the essential meaning of `int(s)`: it takes a string, and returns a number. But the real power of exceptions only appears in the next section…

Exceptions deep deep down The nice thing about exceptions is that you can also catch exceptions that were thrown *inside* functions called in the `try`-block.

Going back to our number conversion example, here is version where we convert the string in a separate function

The function `test(s)` converts the string to a double, but then rounds it off to two decimal places and returns an integer.

When a conversion error occurs, this happens inside `test(s)`, but we can still catch this in the `show(s)` function:

```
def test(s):
  return int(100 * float(s))

def show(s):
  try:
    print(test(s))
  except ValueError:
    print("'%s' is not a number" % s)
```

Here are two calls to `show`:

```
$ python3 -i catch2.py
>>> show("123.456")
12345
>>> show("123a456")
'123a456' is not a number
```

Note that when an exception occurs (in this case inside the `float` function), the `float` function does not return, the `test` function does not return, the `print` statement is not executed, and instead execution continues in the `except`-clause.

In general, when an exception occurs (we say that an exception is *raised*), then the normal flow of execution is then interrupted, and continues at the nearest (*innermost*, most recent) `try`-block where this type of exception is *caught* (that is, there is an *exception handler* of the right type).

Let's look at this in more detail and consider the following program:

```
def f(n):
  print("Starting f(%d) ... " % n)
  g(n)
  print("Ending f(%d) ... " % n)

def g(n):
  print("Starting g(%d) ... " % n)
  m = 100 // n
  print("The result is %d" % m)
  print("Ending g(%d) ... " % n)

def main():
  while True:
    s = input("Enter a number> ")
    if s.strip() == "":
      return
    try:
      print("Beginning of try block")
      n = int(s)
      f(n)
      print("End of try block")
    except ValueError:
      print("Please enter a number!")
    except ZeroDivisionError:
      print("I can't handle this value!")

main()
```

Here is a run:

```
$ python3 except1.py
Enter a number> 12
Beginning of try block
Starting f(12) ...
Starting g(12) ...
The result is 8
Ending g(12) ...
Ending f(12) ...
End of try block
Enter a number> abc
Beginning of try block
Please enter a number!
Enter a number> 0
Beginning of try block
Starting f(0) ...
Starting g(0) ...
I can't handle this value!
```

For input value "12", we see beginning and end of the `try`-block and of the functions `f` and `g`. For input value "abc", the `int(s)` function call throws an exception, so `f` is not called. For input value "0", the division inside function `g` throws a `ZeroDivisionError`. As you see, execution continues *immediately* in the exception handler, without finishing functions `g`, `f`, or the `try`-block.

Raising exceptions So far we have only caught exceptions raised inside some library function. But you can just as well raise exceptions yourself. For instance, let's assume that our function `g(n)` above should only handle non-negative numbers. We can ensure this by throwing a `ValueError` if the argument is negative. The whole script now looks like this:

```
def f(n):
  print("Starting f(%d) ... " % n)
  g(n)
  print("Ending f(%d) ... " % n)

def g(n):
  print("Starting g(%d) ... " % n)
  if n < 0:
    raise ValueError("Cannot handle negative numbers")
  print("The result is %d" % n)
  print("Ending g(%d) ... " % n)

def main():
  while True:
    s = input("Enter a number> ")
    if s.strip() == "":
      return
    try:
      print("Beginning of try block")
      n = int(s)
      f(n)
      print("End of try block")
    except ValueError as e:
      print("Cannot handle this input: %s" % e)

main()
```

The example also shows how we can use the message belonging to the exception inside the exception-handler.

Note that exceptions are *objects*, and are created like any other object, by calling their constructor. The exception message is stored as a field of the exception object.

Again, we run this script with different inputs:

```
$ python3 except2.py
Enter a number> 123
Beginning of try block
Starting f(123) ...
Starting g(123) ...
The result is 123
Ending g(123) ...
Ending f(123) ...
End of try block
Enter a number> abc
Beginning of try block
Cannot handle this input: invalid literal for int() with base 10: 'abc'
Enter a number> -123
Beginning of try block
Starting f(-123) ...
Starting g(-123) ...
Cannot handle this input: Cannot handle negative numbers
```

Exceptions are often used to *detect errors in the input data.*

We can catch the exception at a suitable place in the program and print an error message, or handle the problem in some other way.