Data types and data structures

A data type (also called abstract data type or ADT) defines the operations and behavior supported by an object.

A data type is a concept, similar to mathematical concepts such as function, set, or sequence.

Examples of data types are Stack, Queue, Set, Dictionary.

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.

We often have multiple, different implementations for the same data type: Stacks can be implemented with arrays or with linked lists, sets can be implemented with search trees or with hash tables.

Book definition

An abstract data type is a programmer-defined data type that specifies a set of data values and a collection of well-defined operations that can be performed on those values.

Abstract data types are defined independent of their implementation.

- We can focus on solving the problem instead of the implementation details.
- Reduce logical errors by preventing direct access to the implementation.
- Implementation can be changed.
- Easier to manage and divide larger programs into smaller modules.

Day calculator

Let's build a day calculator for determining the number of days between dates, or the 1000th day after a given day, etc.

```
> 2015/03/20
2015/03/20 is a Friday
> 1992/03/21
1992/03/21 is a Saturday
>  1995/12/01 2015/03/20
There are 7049 days between 1995/12/01 & 2015/03/2(
>  2015/03/20 2014/08/24
There are -208 days between 2015/03/20 & 2014/08/2
>  1995/12/01 + 100
1995/12/01 + 100 days = 1996/03/10
>  2015/03/20 - 1000
2015/03/20 - 1000 days = 2012/06/23
```

ADT for dates

We need an ADT to store a date. We specify it like this:

- `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).

We also want to compare dates, and have a nice string representation.

Client code

Since we have a fully specified `Date` ADT, we can start by writing the client code.

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

Implementing the Date class

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

For the `dayOfWeek`, `numDays`, and `advanceBy` methods we need to convert to and from Julian day number.

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

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

Another implementation?

Imagine an application, where we need to store millions of `Date` objects.

We should make `Date` as small as possible—ideally store only a single number.

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

The methods `year`, `month`, `day` get harder now, but `numDays` and `dayOfWeek` get easier...

```
  def dayOfWeek(self):
    return self._jday % 7
  def numDays(self, otherDate):
    return otherDate._jday - self._jday
```

Equality and Ordering

We want to compare dates using all the standard operators: ==, <, <=, >=, >, !=.

```
  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
```

Invalid dates

Our class does not recognize invalid dates:

```
> 2015/02/29
2015/02/29 is a Sunday
> 2015/09/31
2015/09/31 is a Thursday
> 2015/13/00
2015/13/00 is a Thursday
> 2015/13/01 - 1
2015/13/01 - 1 days = 2015/12/31
> 2015/12/00 + 1
2015/12/00 + 1 days = 2015/12/01
> 2015/02/29 2015/03/01
There are 0 days between 2015/02/29 and 2015/03/01
```

Handling invalid dates

Recognizing invalid dates is easy: convert to day number and back—if it's not equal, it's invalid.

But how to report the error? The `__init__` method must return a `Date` object!

Solution: 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
```

But now the program crashes when we use an invalid date...

Catching exceptions

We need to catch the exception and report the error to the user:

```
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])
  # omitted
except ValueError as e:
  print(e)
```

Exceptions simplify our error handling

We can now also handle the incorrect date formats using exceptions.
This simplifies the entire `days` program: We no longer need to check the result of `get_date` every time we call it.

```
def show_weekday(s):
  day = get_date(s)
  print(day, "is a", dayNames[day.dayOfWeek()])

def show_difference(s1, s2):
  day1 = get_date(s1)
  day2 = get_date(s2)
  print("There are", day1.numDays(day2),
        "days between", day1, "and", day2)
```