



## 2.8 Efficiency

- 2.8.1 Measuring Efficiency
- 2.8.2 Memoization
- 2.8.3 Orders of Growth
- 2.8.4 Example: Exponentiation
- 2.8.5 Growth Categories

## 2.9 Recursive Objects

- 2.9.1 Linked List Class
- 2.9.2 Tree Class
- 2.9.3 Sets

are important to know. In particular, `int` objects represent integers exactly, without any approximation or limits on their size. On the other hand, `float` objects can represent a wide range of fractional numbers, but not all numbers can be represented exactly, and there are minimum and maximum values. Therefore, `float` values should be treated as approximations to real values. These approximations have only a finite amount of precision. Combining `float` values can lead to approximation errors; both of the following expressions would evaluate to 7 if not for approximation.

```
>>> 7 / 3 * 3
7.0
```

```
>>> 1 / 3 * 7 * 3
6.999999999999999
```

Although `int` values are combined above, dividing one `int` by another yields a `float` value: a truncated finite approximation to the actual ratio of the two integers divided.

```
>>> type(1/3)
<class 'float'>
>>> 1/3
0.3333333333333333
```

Problems with this approximation appear when we conduct equality tests.

```
>>> 1/3 == 0.333333333333333312345 # Beware of float approximation
True
```

These subtle differences between the `int` and `float` class have wide-ranging consequences for writing programs, and so they are details that must be memorized by programmers. Fortunately, there are only a handful of native data types, limiting the amount of memorization required to become proficient in a programming language. Moreover, these same details are consistent across many programming languages, enforced by community guidelines such as the [IEEE 754 floating point standard](#).

**Non-numeric types.** Values can represent many other types of data, such as sounds, images, locations, web addresses, network connections, and more. A few are represented by native data types, such as the `bool` class for values `True` and `False`. The type for most values must be defined by programmers using the means of combination and abstraction that we will develop in this chapter.

The following sections introduce more of Python's native data types, focusing on the role they play in creating useful data abstractions. For those interested in further details, a chapter on [native data types](#) in the online book *Dive Into Python 3* gives a pragmatic overview of all Python's native data types and how to manipulate them, including numerous usage examples and practical tips.

*Continue: [2.2 Data Abstraction](#)*