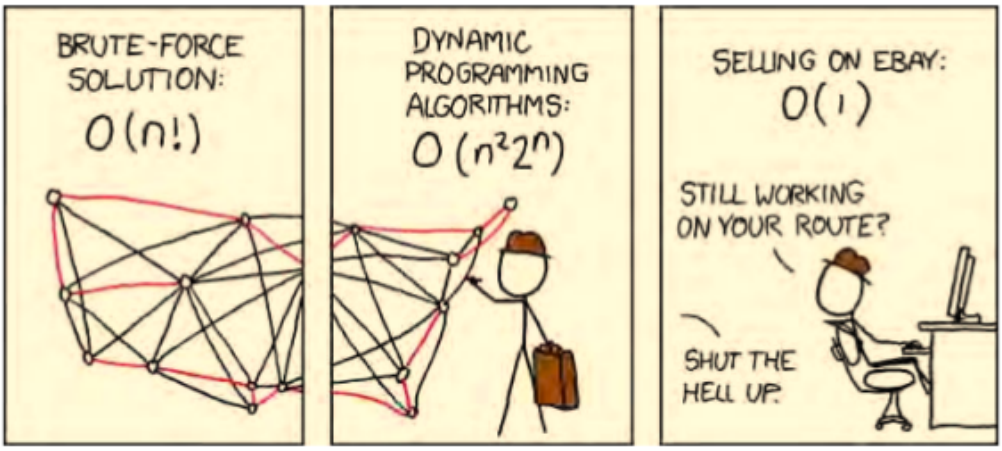
An introduction to Python & Algorithms

Marina von Steinkirch (bt3)
@_b_t_3_

Summer, 2013



"There's nothing to fear but the fear itself.
That's called recursion, and that would lead you to
infinite fear."

Contents

Part I

Get your wings!

Chapter 1

Oh Hay, Numbers!

Numbers can be represented as an `integer`, a `float`, or a `complex` value. Because humans have 10 fingers, we have learned to represent numbers as `decimals`. Computers, however, are made of electrical states, so `binary` representations are more suitable. That's why the represent information with bits. Additionally, representations that are multiples of 2 are equally useful, such as `hexadecimals` and `octals`.

1.1 Integers

Python represents integers using the `immutable int` type. For immutable objects there is no difference between a variable and an *object reference*.

The size of Python's integers is limited by the machine memory (the range depends on the C or Java built-in compiler), being at least 32-bit long (4 bytes)[1].

To see how many bytes is hold in an integer, the `int.bit_length()` method is available (starting in Python 3.):

```
>>> (999).bit_length()
 10
```

To cast a string to an integer (in some base) or to change the base of an integer, we use `int(s, base)`:

[1] To have an idea of how much this means: 1K of disk memory has 1024×8 bits $= 2^{10}$ bytes.

```
>>> s = '11'
>>> d = int(s)
>>> print(d)
11
>>> b = int(s, 2)
>>> print(b)
3
```

The optional base argument must be an integer between 2 and 36 (inclusive). If the string cannot be represented as the integer in the chosen base, this method raises a `ValueError exception`.

1.2 Floats

Numbers with a fractional part are represented by the `immutable` type `float`. When we use *single precision*, a 32-bit float is represented by: **1 bit for sign** (negative being 1, positive being 0), plus **23 bits for the significant digits** (or *mantissa*), plus **8 bits for the exponent**.

Additionally, the exponent is represented using the *biased notation*, where you add the number 127 to the original value[2].

Comparing Floats

Floats have a precision limit, which constantly causes them to be truncated (that's why we should never compare floats for equality!).

Equality tests should be done in terms of some predefined precision. A simple example would be the following function:

```
>>> def a(x , y, places=7):
...     return round(abs(x-y), places) == 0
```

Remarkably, similar approach is used by Python's `unittest` module, with the method `assertAlmostEqual()`.

[2]Biasing is done because exponents have to be signed values to be able to represent tiny and huge values, but the usual representation makes comparison harder. To solve this problem, the exponent is adjusted to be within an unsigned range suitable for comparison. To learn more: http://www.doc.ic.ac.uk/ eedwards/compsys/float

Interestingly, there are several numbers that have an exact representation in a decimal base but not in a binary base (for instance, the decimal 0.1).

Methods for Floats and Integers

In Python, the division operator `/` always returns a float. A `floor` division (truncation) can be made with the operator `//`. A `module` (remainder) operation is given by the operator %.

The method `divmod(x,y)` returns both the quotient and remainder when dividing `x` by `y`:

```
>>> divmod(45,6)
(7, 3)
```

The method `round(x, n)` returns `x` rounded to `n` integral digits if `n` is a negative `int` or returns `x` rounded to `n` decimal places if `n` is a positive `int`:

```
>>> round(100.96,-2)
100.0
>>> round(100.96,2)
100.96
```

The method `as_integer_ratio()` gives the integer fractional representation of a float:

```
>>> 2.75.as_integer_ratio()
(11, 4)
```

1.3 Complex Numbers

A *complex data type* is an `immutable` number that holds a pair of floats (for example, $z = 3 + 4j$). Additional methods are available, such as: `z.real`, `z.imag`, and `z.conjugate()`.

Complex numbers are imported from the `cmath` module, which provides complex number versions for most of the trigonometric and logarithmic functions that are in the math module, plus some complex number-specific functions such as: `cmath.phase()`, `cmath.polar()`, `cmath.rect()`, `cmath.pi`, and `cmath.e`.

1.4 The `fraction` Module

Python has the `fraction` module to deal with parts of a fraction. The following snippet shows the basics methods of this module:[3]

```
[testing_floats.py]

from fractions import Fraction

def rounding_floats(number1, places):
    return round(number1, places)

def float_to_fractions(number):
    return Fraction(number.as_integer_ratio())

def get_denominator(number1, number2):
    a = Fraction(number1, number2)
    return a.denominator

def get_numerator(number1, number2):
    a = Fraction(number1, number2)
    return a.numerator

def test_testing_floats(module_name='this module'):
    number1 = 1.25
    number2 = 1
    number3 = -1
    number4 = 5/4
    number6 = 6
    assert(rounding_floats(number1, number2) == 1.2)
    assert(rounding_floats(number1*10, number3) == 10)
    assert(float_to_fractions(number1) == number4)
    assert(get_denominator(number2, number6) == number6)
    assert(get_numerator(number2, number6) == number2)
```

[3]Every snippet shown in this book is available at https://github.com/bt3gl/Python-and-Algorithms-and-Data-Structures.

1.5 The decimal Module

To handle **exact** decimal floating-point numbers, Python includes an additional `immutable` float type in the `decimal` library. This is an efficient alternative to the rounding, equality, and subtraction problems that `floats` come with:

```
>>> sum (0.1 for i in range(10)) == 1.0
 False
>>> from decimal import Decimal
>>> sum (Decimal ("0.1") for i in range(10)) == Decimal("1.0")
 True
```

While the `math` and `cmath` modules are not suitable for the `decimal` module, it does have these functions built in, such as `decimal.Decimal.exp(x)`.

1.6 Other Representations

The `bin(i)` method returns the binary representation of the `int i`:

```
>>> bin(999)
'0b1111100111'
```

The `hex(i)` method returns the hexadecimal representation of `i`:

```
>>> hex(999)
'0x3e7'
```

The `oct(i)` method returns the octal representation of `i`:

```
>>> oct(999)
'0o1747'
```

1.7 Some Fun Examples

Converting Between Different Bases

We can write our own functions to modify some number's base. The snippet below converts a number in any base smaller than 10 to the decimal base:

```
[convert_to_decimal.py]

def convert_to_decimal(number, base):
    multiplier, result = 1, 0
    while number > 0:
        result += number%10*multiplier
        multiplier *= base
        number = number//10
    return result

def test_convert_to_decimal():
    number, base = 1001, 2
    assert(convert_to_decimal(number, base) == 9)
```

In the previous example, by swapping all the occurrences of 10 with any other `base`, we can create a function that converts from a decimal `number` to another number ($2 \leq$ `base` ≤ 10):

```
[convert_from_decimal.py]

def convert_from_decimal(number, base):
    multiplier, result = 1, 0
    while number > 0:
        result += number%base*multiplier
        multiplier *= 10
        number = number//base
    return result

def test_convert_from_decimal():
    number, base = 9, 2
    assert(convert_from_decimal(number, base) == 1001)
```

If the `base` value is larger than 10 we must use non-numeric characters to represent larger digits. We can let 'A' stand for 10, 'B' stand for 11 and so on. The following code converts a number from a decimal base to any other base (up to 20):

```
[convert_from_decimal_larger_bases.py]

def convert_from_decimal_larger_bases(number, base):
```

```
    strings = "0123456789ABCDEFGHIJ"
    result = ""
    while number > 0:
        digit = number%base
        result = strings[digit] + result
        number = number//base
    return result

def test_convert_from_decimal_larger_bases():
    number, base = 31, 16
    assert(convert_from_decimal_larger_bases(number, base) == '1F')
```

Finally, the snippet below is a **recursive** general base-conversion:

```
[convert_dec_to_any_base_rec.py]

def convert_dec_to_any_base_rec(number, base):
    convertString = '012345679ABCDEF'
    if number < base: return convertString[number]
    else:
        return convert_dec_to_any_base_rec(number//base, base) +
            convertString[number%base]

def test_convert_dec_to_any_base_rec(module_name='this module'):
    number = 9
    base = 2
    assert(convert_dec_to_any_base_rec(number, base) == '1001')
```

Greatest Common Divisor

The following module calculates the **greatest common divisor** (gcd) between two given integers:

```
[finding_gcd.py]

def finding_gcd(a, b):
    while(b != 0):
        result = b
```

```
        a, b = b, a % b
    return result

def test_finding_gcd():
    number1 = 21
    number2 = 12
    assert(finding_gcd(number1, number2) == 3)
    test_finding_gcd()
```

The `Random` Module

The follow snippet runs some tests on the Python's `random` module:

```
[testing_random.py]

import random

def testing_random():
    ''' testing the module random'''
    values = [1, 2, 3, 4]
    print(random.choice(values))
    print(random.choice(values))
    print(random.choice(values))
    print(random.sample(values, 2))
    print(random.sample(values, 3))

    ''' shuffle in place '''
    random.shuffle(values)
    print(values)

    ''' create random integers '''
    print(random.randint(0,10))
    print(random.randint(0,10))
```

Fibonacci Sequences

The module below shows how to find the nth number in a *Fibonacci sequence* in three different ways: (a) with a recursive $\mathcal{O}(2^n)$ runtime; (b) with an iterative $\mathcal{O}(n^2)$ runtime; and (c) using the generator propriety instead:

```
[fibonacci.py]

def fib_generator():
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a+b

def fib(n):
    '''
    >>> fib(2)
    1
    >>> fib(5)
    5
    >>> fib(7)
    13
    '''
    if n < 3:
        return 1
    a, b = 0, 1
    count = 1
    while count < n:
        count += 1
        a, b = b, a+b
    return b

def fib_rec(n):
    '''
    >>> fib_rec(2)
    1
    >>> fib_rec(5)
    5
    >>> fib_rec(7)
    13
    '''
```

```
    if n < 3:
        return 1
    return fib_rec(n - 1) + fib_rec(n - 2)
```

Primes

The following snippet includes functions to find if a number is prime, to give the number's primes factors, and to generate primes:

```
[primes.py]

import math
import random

def find_prime_factors(n):
    '''
    >>> find_prime_factors(14)
    [2, 7]
    >>> find_prime_factors(19)
    []
    '''
    divisors = [d for d in range(2, n//2 + 1) if n % d == 0]
    primes = [d for d in divisors if is_prime(d)]
    return primes

def is_prime(n):
    for j in range(2, int(math.sqrt(n))):
        if (n % j) == 0:
            return False
    return True

def generate_prime(number=3):
    while 1:
        p = random.randint(pow(2, number-2), pow(2, number-1)-1)
        p = 2 * p + 1
        if find_prime_factors(p):
            return p
```

1.8 The NumPy Package

The **NumPy** package[4] is an extension to the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays.

Arrays in NumPy can have any arbitrary dimension. They can be generated from a list or a tuple with the array-method, which transforms sequences of sequences into two dimensional arrays:

```
>>> x = np.array( ((11,12,13), (21,22,23), (31,32,33)) )
>>> print x
[[11 12 13]
 [21 22 23]
 [31 32 33]]
```

The attribute `ndim` tells us the number of dimensions of an array:

```
>>> x = np.array( ((11,12,13), (21,22,23) ))
>>> x.ndim
2
```

Let's see a full test example:

```
[testing_numpy.py]

import numpy as np

def testing_numpy():
    ax = np.array([1,2,3])
    ay = np.array([3,4,5])
    print(ax)
    print(ax*2)
    print(ax+10)
    print(np.sqrt(ax))
    print(np.cos(ax))
    print(ax-ay)
```

[4] http://www.numpy.org

```
    print(np.where(ax<2, ax, 10))

    m = np.matrix([ax, ay, ax])
    print(m)
    print(m.T)

    grid1 = np.zeros(shape=(10,10), dtype=float)
    grid2 = np.ones(shape=(10,10), dtype=float)
    print(grid1)
    print(grid2)
    print(grid1[1]+10)
    print(grid2[:,2]*2)
```

NumPy arrays are also much more efficient than Python's lists, as we can see in the benchmark tests below:

```
[testing_numpy_speed.py]

import numpy
import time

def trad_version():
    t1 = time.time()
    X = range(10000000)
    Y = range(10000000)
    Z = []
    for i in range(len(X)):
        Z.append(X[i] + Y[i])
    return time.time() - t1

def numpy_version():
    t1 = time.time()
    X = numpy.arange(10000000)
    Y = numpy.arange(10000000)
    Z = X + Y
    return time.time() - t1

if __name__ == '__main__':
    assert(trad_version() > 3.0)
    assert(numpy_version() < 0.1)
```

Chapter 2

Built-in Sequence Types

In this chapter we will learn how Python represents `sequence data types`. A sequence type is defined by having the following properties:

- ⋆ **membership operator** (for example, the ability of using the keyword `in`);
- ⋆ **a size method** (given by the method `len(seq)`);
- ⋆ **a slicing properties** (for example, `seq[:-1]`); and
- ⋆ **iterability** (data can be used inside loops).

Python has five built-in sequence types: **strings**, **tuples**, **lists**, **byte arrays**, and **bytes**:[1]

```
>>> l = []
>>> type(l)
<type 'list'>
>>> s = ''
>>> type(s)
<type 'str'>
>>> t = ()
>>> type(t)
<type 'tuple'>
>>> ba = bytearray(b'')
```

[1] A **named tuple** is also available in the standard library, under the `collections` package.

```
>>> type(ba)
<type 'bytearray'>
>>> b = bytes([])
>>> type(byte)
<type 'type'>
```

Mutability

In Python, tuple, strings, and bytes are `immutable`, while lists and byte arrays are `mutable`.

Immutable types are in general more efficient than mutable objects. In addition, some *collection data types*[2] can only be indexed by immutable data types.

In Python, any variable is an *object reference*, so copying mutable objects can be tricky. When you say $a = b$ you are actually pointing a to where b points to. For this reason, it's important to understand the concept of `deep copying`.

For instance, to make a deep copy of a list, we do:

```
>>> newList = myList[:]
>>> newList2 = list(myList2)
```

To make a deep copy of a set, we do:

```
>>> people = {"Buffy", "Angel", "Giles"}
>>> slayers = people.copy()
>>> slayers.discard("Giles")
>>> slayers.remove("Angel")
>>> slayers
{'Buffy'}
>>> people
{'Giles', 'Buffy', 'Angel'}
```

To make a deep copy of a dict, we do:

```
>>> newDict = myDict.copy()
```

To make a deep copy of some other object, we use the copy module:

[2] Collection data types, such as sets and dictionaries, are reviewed in the next chapter.

```
>>> import copy
>>> newObj = copy.copy(myObj)     # shallow copy
>>> newObj2 = copy.deepcopy(myObj2) # deep copy
```

The Slicing Operator

In Python's sequence types, the slicing operator have the following syntax:

```
seq[start]
seq[start:end]
seq[start:end:step]
```

If we want to start counting from the right, we can represent the index as negative:

```
>>> word = "Let us kill some vampires!"
>>> word[-1]
'!'
>>> word[-2]
's'
>>> word[-2:]
's!'
>>> word[:-2]
'Let us kill some vampire'
>>> word[-0]
'L'
```

2.1 Strings

In Python, every object has two output forms: while *string forms* are designed to be human-readable, *representational forms* are designed to produce an output that if fed to a Python interpreter, reproduces the represented object.

Python represents `strings`, *i.e.* a sequence of characters, with the `immutable str` type.

Unicode Strings

Python's *Unicode encoding* is used to include special characters in strings (for example, *whitespaces*). Additionally, starting from Python 3, all strings are Unicode, not just plain bytes. The difference is that, while ASCII representations are given by 8-bits, Unicode representations usually use 16-bits.

To create an Unicode string, we use the 'u' prefix:

```
>>> u'Goodbye\u0020World !'
'Goodbye World !'
```

In the example above, the escape sequence indicates the Unicode character with the ordinal value `0x0020`.

Methods for Strings

The `join(list1)` Method:

Joins all the strings in a list into one single string. While we could use `+` to concatenate these strings, when a large volume of data is involved, this becomes much less efficient:

```
>>> slayer = ["Buffy", "Anne", "Summers"]
>>> " ".join(slayer)
'Buffy Anne Summers'
>>> "-<>-".join(slayer)
'Buffy-<>-Anne-<>-Summers'
>>> "".join(slayer)
'BuffyAnneSummers'
```

`join()` can also be used with the built-in `reversed()` method:

```
>>> "".join(reversed(slayer))
'SummersAnneBuffy'
```

The `rjust(width[, fillchar])` and `ljust(width[, fillchar])` Methods:

Some formation (aligning) can be obtained with the methods `rjust()` (add only at the end), `ljust()` (add only at the start):

```
>>> name = "Agent Mulder"
>>> name.rjust(50, '-')
'----------------------------Agent Mulder'
```

The `format()` Method:

Used to format or add variable values to a string:

```
>>> "{0} {1}".format("I'm the One!", "I'm not")
"I'm the One! I'm not"
>>> "{who} turned {age} this year!".format(who="Buffy", age=17)
'She turned 88 this year'
>>> "The {who} was {0} last week".format(12, who="boy")
'Buffy turned 17 this year!'
```

Starting from Python 3.1, it is possible to omit field names:

```
>>> "{} {} {}".format("Python", "can", "count")
'Python can count'
```

This method allows three specifiers: `s` to force string form, `r` to force representational form, and `a` to force representational form but only using ASCII characters:

```
>>> import decimal
>>> "{0} {0!s} {0!r} {0!a}".format(decimal.Decimal("99.9"))
"99.9 99.9 Decimal('99.9') Decimal('99.9')"
```

String (Mapping) Unpacking

The mapping unpacking operator is `**` and it produces a key-value list suitable for passing to a function. The local variables that are currently in scope are available from the built-in `locals()` and this can be used to feed the `format()` method:

```
>>> hero = "Buffy"
>>> number = 999
>>> "Element {number} is a {hero}".format(**locals())
'Element 999 is a Buffy'
```

The `splitlines(f)` Method:

Returns the list of lines produced by splitting the string on line terminators, stripping the terminators unless `f` is True:

```
>>> slayers = "Buffy\nFaith"
>>> slayers.splitlines()
['Buffy', 'Faith']
```

The `split(t, n)` Method:

Returns a list of strings splitting at most `n` times on string `t`. If `n` is not given, it splits as many times as possible. If `t` is not given, it splits on whitespace:

```
>>> slayers = "Buffy*Slaying-Vamps*16"
>>> fields = slayers.split("*")
>>> fields
['Buffy', 'Slaying-Vamps', '16']
>>> job = fields[1].split("-")
>>> job
['Slaying', 'Vamps']
```

We can use `split()` to write our own method for erasing spaces from strings:

```
>>> def erase_space_from_string(string):
...     s1 = string.split(" ")
...     s2 = "".join(s1)
...     return s2
```

A similar method, `rsplit()`, splits the string from right to left.

The `strip('chars')` Method:

Returns a copy of the string with leading and trailing whitespace (or the characters `chars`) removed:

```
>>> slayers = "Buffy and Faith999"
```

```
>>> slayers.strip("999")
'Buffy and Faith'
```

The program below uses `strip()` to list every word and the number of the times they occur in alphabetical order for some file:[3]

```
[count_unique_words.py]

import string
import sys

def count_unique_word():
    words = {} # create an empty dictionary
    strip = string.whitespace + string.punctuation + string.digits
        + "\"'"
    for filename in sys.argv[1:]:
        with open(filename) as file:
            for line in file:
                for word in line.lower().split():
                    word = word.strip(strip)
                    if len(word) > 2:
                        words[word] = words.get(word,0) +1
    for word in sorted(words):
        print("'{0}' occurs {1} times.".format(word, words[word]))
```

Similar methods are: `lstrip()`, which returns a copy of the string with all whitespace at the beginning of the string stripped away; and `rstrip()`, which returns a copy of the string with all whitespace at the end of the string stripped away.

The `swapcase('chars')` Method:

The `swapcase()` method returns a copy of the string with uppercase characters lowercased and vice-versa:

```
>>> slayers = "Buffy and Faith"
>>> slayers.swapcase()
'bUFFY AND fAITH'
```

[3] A similar example is shown in the Default Dictionaries section.

Similar methods are:

- ⋆ `capitalize()`: returns a copy of the string with only the first character in uppercase;
- ⋆ `lower()`: returns a copy of the original string, but with all characters in lowercase;
- ⋆ `upper()`: returns a copy of the original string, but with all characters in uppercase.

The `index(x)` and `find(x)` Methods:

There are two methods to find the position of a string inside another string. `index(x)` returns the index position of the substring `x`, or raises a `ValueError` exception on failure. `find(x)` returns the index position of the substring `x`, or -1 on failure:

```
>>> slayers = "Buffy and Faith"
>>> slayers.find("y")
4
>>> slayers.find("k")
-1
>>> slayers.index("k")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> slayers.index("y")
4
```

Extensions of the previous methods are: `rfind(string)`, which returns the index within the string of the last (from the right) occurrence of 'string'; and `rindex(string)`, which returns the index within the string of the last (from the right) occurrence of 'string' (causing an error if it cannot be found).

The `count(t, start, end)` Method:

Returns the number of occurrences of the string `t` in the string `s`:

```
>>> slayer = "Buffy is Buffy is Buffy"
>>> slayer.count("Buffy", 0, -1)
```

```
2
>>> slayer.count("Buffy")
3
```

The `replace(t, u, n)` Method:

Returns a copy of the string with every (or a maximum of `n` if given) occurrences of string `t` replaced with string `u`:

```
>>> slayer = "Buffy is Buffy is Buffy"
>>> slayer.replace("Buffy", "who", 2)
'who is who is Buffy'
```

2.2 Tuples

A **tuple** is an Python `immutable` sequence type consisting of values separated by commas:

```
>>> t1 = 1234, 'hello!'
>>> t1[0]
1234
>>> t1
(12345, 'hello!')
>>> t2 = t2, (1, 2, 3, 4, 5) # nested
>>> u
((1234, 'hello!'), (1, 2, 3, 4, 5))
```

Where strings have a character at every position, tuples have an *object reference* at each position. For this reason, it is possible to create tuples that contain mutable objects, such as lists.

Empty tuples are constructed by an empty pair of parentheses, and tuples with one item are constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses):

```
>>> empty = ()
>>> t1 = 'hello',
>>> len(empty)
```

```
0
>>> len(t1)
1
>>> t1
('hello',)
```

Methods for Tuples

The `count(x)` method counts how many times `x` appears in the tuple:

```
>>> t = 1, 5, 7, 8, 9, 4, 1, 4
>>> t.count(4)
2
```

The `index(x)` method returns the index position of the element `x`:

```
>>> t = 1, 5, 7
>>> t.index(5)
1
```

Tuple Unpacking

In Python, any iterable object can be unpacked using the *sequence unpacking operator*, `*`. This operator can be used with two or more variables. In this case, the items in the left-hand side of the assignment (which are preceded by `*`) are assigned to the named variables, while the items left over are assigned to the starred variable:

```
>>> x, *y = (1, 2, 3, 4)
>>> x
1
>>> y
[2, 3, 4]
```

Named Tuples

Python's package `collections`[4] contains a sequence data type called `named tuple`. These objects behave just like the built-in tuple, with the same performance characteristics, but in addition they carry the ability to refer to items by name. This allows the creation of aggregates of data items:

```
>>> import collections
>>> MonsterTuple = collections.namedtuple("Monsters","name age
    power")
>>> MonsterTuple = ('Vampire', 230, 'immortal')
>>> MonsterTuple
('Vampire', 230, 'immortal')
```

The first argument to `collections.namedtuple` is the name of the custom tuple data type to be created. The second argument is a string of space-separated names, one for each item that the custom tuple will take. The first argument and the names in the second argument must be valid Python identifiers:

```
[namedtuple_example.py]

from collections import namedtuple

def  namedtuple_example():
    >>> namedtuple_example()
    slayer
    '''
    sunnydale = namedtuple('name', ['job', 'age'])
    buffy = sunnydale('slayer', '17')
    print(buffy.job)
```

2.3 Lists

In Computer Science, *arrays* are a very simple data structure where elements are sequentially stored in continued memory, and *linked lists* are structures

[4] We are going to explore `collections` in the following chapters.

where several separated nodes link to each other. Iterating over the contents is equally efficient for both kinds, but *directly accessing* an element at a given index has $\mathcal{O}(1)$ (complexity) runtime[5] in an array, while it has $\mathcal{O}(n)$ in a linked list with n nodes (you would have to transverse the list from the beginning).

Additionally, in a linked list, once you know where you want to insert something, *insertion* is $\mathcal{O}(1)$, no matter how many elements the list has. For arrays, an insertion would have to move all elements that are to the right of the insertion point or moving all the elements to a larger array if needed, being then $\mathcal{O}(n)$.

In Python, the closest object to an array is a **list**, which is a dynamic resizing array and it does not have anything to do with the formal concept of linked lists. Linked lists are a very important *abstract data structure* (we will see more about them in a following chapter) and it is fundamental to understand what makes them different from arrays (or Python's lists).

Lists in Python are created by comma-separated values, between square brackets. Items in lists do not need to have all the same data type. Unlike strings which are immutable, it is possible to change individual elements of a list (lists are `mutable`):

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> p[1].append("buffy")
>>> p
[1, [2, 3, 'buffy'], 4]
>>> q
[2, 3, 'buffy']
>>> q
[2, 3, 'buffy']
```

To insert items, lists perform best ($\mathcal{O}(1)$) when items are added or removed at the end, using the methods `append()` and `pop()`, respectively. The worst performance ($\mathcal{O}(n)$) occurs with operations that need to search for items in the list, for example, using `remove()` or `index()`, or using `in` for membership testing.[6]

[5] The Big-O notation is a key to understand algorithms. We will learn it in the following chapters. For now, keep in mine that $\mathcal{O}(1) \ll \mathcal{O}(n) \ll \mathcal{O}(n^2)$, etc...

[6] This explains why `append()` is so much more efficient than `insert()`.

If fast searching or membership testing is required, a collection type such as a *set* or a *dictionary* may be a more suitable choice (as we will see in the next chapter). Alternatively, lists can provide fast searching if they are kept in order by being sorted (we will see searching methods that perform on $\mathcal{O}(\log n)$ for sorted sequences, particularly the *binary search*, in the following chapters).

Methods for Lists

The `append(x)` Method:

Adds a new element at the end of the list. It is equivalent to `list[len(list):]=[x]`:

```
>>> people = ["Buffy", "Faith"]
>>> people.append("Giles")
>>> people
['Buffy', 'Faith', 'Giles']
>>> people[len(people):] = ["Xander"]
>>> people
['Buffy', 'Faith', 'Giles', 'Xander']
```

The `extend(c)` Method:

This method is used to extend the list by appending all the iterable items in the given list. Equivalent to `a[len(a):]=L` or using `+=`:

```
>>> people = ["Buffy", "Faith"]
>>> people.extend("Giles")
>>> people
['Buffy', 'Faith', 'G', 'i', 'l', 'e', 's']
>>> people += "Willow"
>>> people
['Buffy', 'Faith', 'G', 'i', 'l', 'e', 's', 'W', 'i', 'l', 'l',
   'o', 'w']
>>> people += ["Xander"]
>>> people
['Buffy', 'Faith', 'G', 'i', 'l', 'e', 's', 'W', 'i', 'l', 'l',
   'o', 'w', 'Xander']
```

The `insert(i, x)` Method:

Inserts an item in a given position `i`: the first argument is the index of the element *before which* to insert:

```
>>> people = ["Buffy", "Faith"]
>>> people.insert(1, "Xander")
>>> people
['Buffy', 'Xander', 'Faith']
```

The `remove()` Method:

Removes the first item from the list whose value is `x`. Raises a `ValueError` exception if `x` is not found:

```
>>> people = ["Buffy", "Faith"]
>>> people.remove("Buffy")
>>> people
['Faith']
>>> people.remove("Buffy")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

The `pop()` Method:

Removes the item at the given position in the list, and then returns it. If no index is specified, `pop()` returns the last item in the list:

```
>>> people = ["Buffy", "Faith"]
>>> people.pop()
'Faith'
>>> people
['Buffy']
```

The `del`:

Deletes the object reference, not the content, *i.e.*, it is a way to remove an item from a list given its index instead of its value. This can also be used to remove slices from a list:

```
>>> a = [-1, 4, 5, 7, 10]
>>> del a[0]
>>> a
[4, 5, 7, 10]
>>> del a[2:3]
>>> a
[4, 5, 10]
>>> del a     # also used to delete entire variable
```

When an object reference is deleted and no other object refers to its data, Python schedules the item to be *garbage-collected*.[7]

The `index(x)` Method:

Returns the index in the list of the first item whose value is `x`:

```
>>> people = ["Buffy", "Faith"]
>>> people.index("Buffy")
0
```

The `count(x)` Method:

Returns the number of times `x` appears in the list:

```
>>> people = ["Buffy", "Faith", "Buffy"]
>>> people.count("Buffy")
2
```

[7]Garbage is a memory occupied by objects that are no longer referenced and garbage collection is a form of automatic memory management, freeing the memory occupied by the garbage.

The `sort()` Method:

Sorts the items of the list, in place:

```
>>> people = ["Xander", "Faith", "Buffy"]
>>> people.sort()
>>> people
['Buffy', 'Faith', 'Xander']
```

The `reverse()` Method:

Reverses the elements of the list, in place:

```
>>> people = ["Xander", "Faith", "Buffy"]
>>> people.reverse()
>>> people
['Buffy', 'Faith', 'Xander']
```

List Unpacking

Uunpacking in lists is similar to tuples:

```
>>> first, *rest = [1,2,3,4,5]
>>> first
1
>>> rest
[2, 3, 4, 5]
```

Python also has a related concept named *starred arguments*, that can be used as a passing argument for a function:

```
>>> def example_args(a, b, c):
...     return a * b * c  # here * is the multiplication operator
>>> L = [2, 3, 4]
>>> example_args(*L)
24
>>> example_args(2, *L[1:])
24
```

List Comprehensions

A *list comprehension* is an expression and loop (with an optional condition) enclosed in brackets:

```
[item for item in iterable]
[expression for item in iterable]
[expression for item in iterable if condition]
```

Below, some examples of list comprehensions:

```
>>> a = [y for y in range(1900, 1940) if y%4 == 0]
>>> a
[1900, 1904, 1908, 1912, 1916, 1920, 1924, 1928, 1932, 1936]
>>> b = [2**i for i in range(13)]
>>> b
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
>>> c = [x for x in a if x%2==0]
>>> c
[0, 4, 16, 36, 64]
>>> d = [str(round(355/113.0,i)) for i in range(1,6)]
>>> d
['3.1', '3.14', '3.142', '3.1416', '3.14159']
>>> words = 'Buffy is awesome and a vampire slayer'.split()
>>> e = [[w.upper(), w.lower(), len(w)] for w in words]
>>> for i in e:
...     print(i)
['BUFFY', 'buffy', 5]
['IS', 'is', 2]
['AWESOME', 'awesome', 7]
['AND', 'and', 3]
['A', 'a', 1]
['VAMPIRE', 'vampire', 7]
['SLAYER', 'slayer', 6]
```

List comprehensions should only be used for simple cases, when each portion fits in one line (no multiple `for` clauses or `filter` expressions):

```
[Good]
```

```
  result = []
  for x in range(10):
      for y in range(5):
          if x * y > 10:
              result.append((x, y))

  for x in range(5):
      for y in range(5):
          if x != y:
              for z in range(5):
                  if y != z:
                      yield (x, y, z)

  return ((x, complicated_transform(x))
          for x in long_generator_function(parameter)
          if x is not None)

  squares = [x * x for x in range(10)]

  eat(jelly_bean for jelly_bean in jelly_beans
      if jelly_bean.color == 'black')

[Bad]
  result = [(x, y) for x in range(10) for y in range(5) if x * y >
      10]

  return ((x, y, z)
          for x in xrange(5)
          for y in xrange(5)
          if x != y
          for z in xrange(5)
          if y != z)
```

Runtime Analysis for Lists

To understand the performance of Python's lists, we can benchmark some lists' methods.

In the snippet below, we use Python's `timeit` module to create a `Timer` object whose first parameter is what we want to time and the second param-

eter is a statement to set up the test. The `timeit` module will time how long it takes to execute the statement some number of times (one million times by default).

When the test is done, the function returns the time as a floating point value representing the total number of seconds:

```
[runtime_lists_with_timeit_module.py]

def test1():
    l = []
    for i in range(1000):
        l = l + [i]
def test2():
    l = []
    for i in range(1000):
        l.append(i)

def test3():
    l = [i for i in range(1000)]

def test4():
    l = list(range(1000))

if __name__ == '__main__':
    import timeit
    t1 = timeit.Timer("test1()", "from __main__ import test1")
    print("concat ",t1.timeit(number=1000), "milliseconds")
    t2 = timeit.Timer("test2()", "from __main__ import test2")
    print("append ",t2.timeit(number=1000), "milliseconds")
    t3 = timeit.Timer("test3()", "from __main__ import test3")
    print("comprehension ",t3.timeit(number=1000), "milliseconds")
    t4 = timeit.Timer("test4()", "from __main__ import test4")
    print("list range ",t4.timeit(number=1000), "milliseconds")
```

One of the results of this snippet was:

```
('concat ', 2.366791009902954, 'milliseconds')
('append ', 0.16743111610412598, 'milliseconds')
('comprehension ', 0.06446194648742676, 'milliseconds')
('list range ', 0.021029949188232422, 'milliseconds')
```

So we see the following pattern for lists:

```
Operation       Big-O Efficiency
index []       O(1)
index assignment O(1)
append            O(1)
pop()           O(1)
pop(i)            O(n)
insert(i,item)   O(n)
del operator     O(n)
iteration         O(n)
contains (in)    O(n)
get slice [x:y]     O(k)
del slice         O(n)
set slice         O(n+k)
reverse           O(n)
concatenate       O(k)
sort            O(n log n)
multiply       O(nk)
```

2.4 Bytes and Byte Arrays

Python provides two data types for handling raw bytes: `bytes` which is `immutable`, and `bytearray`, which is `mutable`. Both types hold a sequence of zero or more unsigned 8-bits integers in the range 0 ... 255. The `byte` type is very similar to the `string` type and the `bytearray` provides mutating methods similar to `lists`.

Bits and Bitwise Operations

Bitwise operations can be very useful to manipulate numbers represented as bits. For example, we can quickly compute 2^x by the *left-shifting* operation: $1 \gg x$. We can also quickly verify whether a number is a power of 2 by checking whether $x\&(x-1)$ is 0 (if x is not an even power of 2, the highest position of x with a 1 also has an 1 in $x-1$, otherwise, x will be 100...0 and $x-1$ will be 011...1; add them together they return 0).

2.5 Further Examples

Reversing Strings

Python makes it really simple to revert strings:

```
[reversing_strings.py]

def revert(string):
    '''
    >>> s = 'hello'
    >>> revert(s)
    'olleh'
    >>> revert('')
    ''
    '''
    return string[::-1]

def reverse_string_inplace(s):
    '''
    >>> s = 'hello'
    >>> reverse_string_inplace(s)
    'olleh'
    >>> reverse_string_inplace('')
    ''
    '''
    if s:
        s = s[-1] + reverse_string_inplace(s[:-1])
    return s
```

Reversing Words in a String, without Reversing the Words

We want to invert the words in a string, without reverting the words. It is important to remember that Python strings are immutable, so we might want to convert the strings to lists in some cases.

There are many ways we can solve this problems. We can use Python's builtin methods for lists and strings, or we can work with pointers. In the second case, the solution consists of two loops. The first loop will revert all

the characters utilizing two pointers. The second loop will search for spaces and then revert the words.

Further caveats of this problems is that we can represent space as ' ' or as Unicodes (u0020). We also might have to pay attention in the last word, since it does not end with a space. Finally, an extension of this problem is to look for symbols such as `!?;-.`":

```
[reversing_words.py]

def reversing_words(word):
    """
    >>> reversing_words('buffy is awesome')
    'awesome is buffy'
    """
    new_word = []
    words = word.split(' ')
    for word in words[::-1]:
        new_word.append(word)
    return " ".join(new_word)

def reversing_words2(s):
    """
    >>> reversing_words2('buffy is awesome')
    'awesome is buffy'
    """
    words = s.split()
    return ' '.join(reversed(words))

def reversing_words3(s):
    """
    >>> reversing_words('buffy is awesome')
    'awesome is buffy'
    """
    words = s.split(' ')
    words.reverse()
    return ' '.join(words)
```

Simple String Compression

```
[simple_str_compression.py]

from collections import Counter

def str_comp(s):
    '''
    >>> s1 = 'aabcccccaaa'
    >>> str_comp(s1)
    'a2b1c5a3'
    >>> str_comp('')
    ''
    '''
    count, last = 1, ''
    list_aux = []
    for i, c in enumerate(s):
        if last == c:
            count += 1
        else:
            if i != 0:
                list_aux.append(str(count))
            list_aux.append(c)
            count = 1
            last = c
    list_aux.append(str(count))
    return ''.join(list_aux)
```

String Permutation

```
[permutation.py]

def perm(str1):
    '''
    >>> perm('123')
    ['123', '132', '231', '213', '312', '321']
    '''
    if len(str1) < 2:
        return str1
    res = []
```

```
    for i, c in enumerate(str1):
        for cc in perm(str1[i+1:] + str1[:i]):
            res.append(c + cc)
    return res
```

String Combination

```
[combination.py]

def combinations(s):
    '''
    >>> combinations('abc')
    ['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
    >>> combinations('')
    ''
    '''
    if len(s) < 2:
        return s
    res = []
    for i, c in enumerate(s):
        res.append(c)
        for j in combinations(s[:i] + s[i+1:]):
            res.append(c + j)
    return res
```

Palindrome

```
[palindrome.py]

from collections import defaultdict

def is_palindrome(array):
    '''
    >>> is_palindrome('subi no onibus')
    True
    >>> is_palindrome('helllo there')
    False
```

```
    >>> is_palindrome('h')
    True
    >>> is_palindrome('')
    True
    '''
    array = array.strip(' ')
    if len(array) < 2:
        return True
    if array[0] == array[-1]:
        return is_palindrome(array[1:-1])
    else:
        return False
```

Chapter 3

Collection Data Structures

Differently from sequence structures, where the data can be ordered or sliced, `collection` structures are containers that aggregates data without relating them. They have the following proprieties:

- ⋆ **membership operator** (for example, using `in`);
- ⋆ **a size method** (given by `len(seq)`); and
- ⋆ **iterability** (suitable for loops).

In Python, built-in collection types are given by `sets` and `dicts`. Additional collection types are found in the `collections` package, as we discuss in end of this chapter.

3.1 Sets

A `Set` is an unordered collection data type that is iterable, mutable, and has no duplicate elements. Sets are used for *membership testing* and *eliminating duplicate entries*.

Since sets have $\mathcal{O}(1)$ *insertion*, the runtime of *union* is $\mathcal{O}(m+n)$. For *intersection*, it is only necessary to transverse the smaller set, so the runtime is $\mathcal{O}(n)$. [1]

[1] Python's `collection` package has supporting for *Ordered sets*. This data type enforces some predefined comparison for their members.

Frozen Sets

Frozen sets are immutable objects that only support methods and operators that produce a result without affecting set to which they are applied.

Methods for Sets

The `add(x)` Method:

Adds the item `x` to set if it is not already in:

```
>>> people = {"Buffy", "Angel", "Giles"}
>>> people.add("Willow")
>>> people
{'Willow', 'Giles', 'Buffy', 'Angel'}
```

The `s.update(t)` or $s| = t$ Methods:

They both return a set s with elements added from t.

The `s.union(t)` or $s|t$ Methods:

Perform union of the two sets.

The `s.intersection(t)` or $s\&t$ Methods:

Return a new set that has common items from the sets:

```
>>> people = {"Buffy", "Angel", "Giles", "Xander"}
>>> people.intersection({"Angel", "Giles", "Willow"})
{'Giles', 'Angel'}
```

The `s.difference(t)` or $s - t$ Methods:

Return a new set that has every item that is not in the second set:

```
>>> people = {"Buffy", "Angel", "Giles", "Xander"}
>>> vampires = {"Spike", "Angel", "Drusilia"}
>>> people.difference(vampires)
```

```
{'Xander', 'Giles', 'Buffy'}
```

The `clear()` Method:

Removes all the items in the set:

```
>>> people = {"Buffy", "Angel", "Giles"}
>>> people.clear()
>>> people
set()
```

The `discard(x)`, `remove(x)`, and `pop()` Methods:

`discard(x)` removes the item `x` from the set. `remove(x)` removes the item `x` from the set or raises a `KeyError` exception if the element is not in the set. `pop()` returns and removes a random item from the set or raises a `KeyError` exception if the set is empty.

Sets with Lists and Dictionaries

You can cast a set from a list. The snippet below shows some of the available set operations on lists:

```
[set_operations_with_lists.py]

def difference(l1):
    """ return the list with duplicate elements removed """
    return list(set(l1))

def intersection(l1, l2):
    """ return the intersection of two lists """
    return list(set(l1) & set(l2))

def union(l1, l2):
    """ return the union of two lists """
    return list(set(l1) | set(l2))

def test_sets_operations_with_lists():
    l1 = [1,2,3,4,5,9,11,15]
```

```
    l2 = [4,5,6,7,8]
    l3 = []
    assert(difference(l1) == [1, 2, 3, 4, 5, 9, 11, 15])
    assert(difference(l2) == [8, 4, 5, 6, 7])
    assert(intersection(l1, l2) == [4,5])
    assert(union(l1, l2) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15])
    assert(difference(l3) == [])
    assert(intersection(l3, l2) == l3)
    assert(sorted(union(l3, l2)) == sorted(l2))
    print('Tests passed!')

if __name__ == '__main__':
    test_sets_operations_with_lists()
```

We can also use sets' proprieties in dictionaries:[2]

```
[set_operations_dict.py]

from collections import OrderedDict

def  set_operations_with_dict():
    pairs = [('a', 1), ('b',2), ('c',3)]
    d1 = OrderedDict(pairs)
    print(d1)  # ('a', 1), ('b', 2), ('c', 3)

    d2 = {'a':1, 'c':2, 'd':3, 'e':4}
    print(d2)  # {'a': 1, 'c': 2, 'e': 4, 'd': 3}

    union = d1.keys() & d2.keys()
    print(union)  # {'a', 'c'}

    union_items = d1.items() & d2.items()
    print(union_items) # {('a', 1)}

    subtraction1 = d1.keys() - d2.keys()
    print(subtraction1) # {'b'}
```

[2]Sets properties can be used on the dict's attributes `items()` and `keys()` attributes, however `values()` do not support set operations.

```
    subtraction2 = d2.keys() - d1.keys()
    print(subtraction2) # {'d', 'e'}

    subtraction_items = d1.items() - d2.items()
    print(subtraction_items) # {('b', 2), ('c', 3)}

    ''' we can remove keys from a dict doing: '''
    d3 = {key:d2[key] for key in d2.keys() - {'c', 'd'}}
    print(d3) {'a': 1, 'e': 4}

if __name__ == '__main__':
    set_operations_with_dict()
```

3.2 Dictionaries

Dictionaries in Python are implemented using *hash tables*[3]. Hashing functions assign integer values to an arbitrary object in constant time, which can be used as an index into an array:

```
>>> hash(42)
 42
>>> hash("hello")
 355070280260770553
```

A `dict` is a `collection mapping` type that is iterable and supports the membership operator `in` and the size function `len()`. When iterated, unordered mapping types provide their items in an arbitrary order.

Accessing dictionaries has runtime $\mathcal{O}(1)$ so they are used to keep counts of unique items (for example, counting the number of each unique word in a file) and for fast membership test:

```
>>> tarantino = {}
>>> tarantino['name'] = 'Quentin Tarantino'
>>> tarantino['job'] = 'director'
>>> tarantino
```

[3]A hash table is a data structure used to implement an associative array, a structure that can map keys to values.

```
{'job': 'director', 'name': 'Quentin Tarantino'}
>>>
>>> sunnydale = dict({"name":"Buffy", "age":16, "hobby":"slaying"})
>>> sunnydale
{'hobby': 'slaying', 'age': 16, 'name': 'Buffy'}
>>>
>>> sunnydale = dict(name="Giles", age=45, hobby="Watch")
>>> sunnydale
{'hobby': 'Watch', 'age': 45, 'name': 'Giles'}
>>>
>>> sunnydale = dict([("name", "Willow"), ("age",15), ("hobby",
    "nerding")])
>>> sunnydale
{'hobby': 'nerding', 'age': 15, 'name': 'Willow'}
```

Methods for Dictionaries

The `setdefault(key[, default])` Method:

The `setdefault()` method is used when we want to access a key in the dictionary without being sure that this key exists (if we simply try to access a non-existent key in a dictionary, we will get an exception). With `setdefault()`, if key is in the dictionary, we get the value to it. If not, we successfully insert the new key with the value of default:

```
[example_setdefault.py]

def usual_dict(dict_data):
    newdata = {}
    for k, v in dict_data:
        if k in newdata:
            newdata[k].append(v)
        else:
            newdata[k] = [v]
    return newdata

def setdefault_dict(dict_data):
    newdata = {}
```

```
    for k, v in dict_data:
        newdata.setdefault(k, []).append(v)
    return newdata

def test_setdef(module_name='this module'):
    dict_data = (('key1', 'value1'),
                 ('key1', 'value2'),
                 ('key2', 'value3'),
                 ('key2', 'value4'),
                 ('key2', 'value5'),)
    print(usual_dict(dict_data))
    print(setdefault_dict(dict_data))
```

The `update([other])` Method:

Updates the dictionary with the key/value pairs from other, overwriting existing keys.

The `get(key)` Method:

Returns the key's associated value or `None` if the key is not in the dictionary:

```
>>> sunnydale = dict(name="Xander", age=17, hobby="winning")
>>> sunnydale.get("hobby")
'winning'
```

The `items()`, `values()`, and `keys()` Methods:

The `items()`, `keys()`, and `values()` methods all return dictionary views. A dictionary view is effectively a read-only iterable object that appears to hold the dictionary's items or keys or values:

```
>>> sunnydale = dict(name="Xander", age=17, hobby="winning")
>>> sunnydale.items()
dict_items([('hobby', 'winning'), ('age', 17), ('name', 'Xander')])
>>> sunnydale.values()
dict_values(['winning', 17, 'Xander'])
>>> sunnydale.keys()
dict_keys(['hobby', 'age', 'name'])
```

The `pop()` and `popitem()` Methods:

The `pop()` method removes an arbitrary item from the dictionary, returning it. The `popitem()` method removes an arbitrary (key, value) from the dictionary, also returning it.

The `clear()` Method:

Removes all the items in the dictionary:

```
>>> sunnydale.clear()
>>> sunnydale
{}
```

Runtime Analysis for Dictionaries

We can analyze Python's dictionary performance by benchmarking some available methods:

```
[runtime_dicts_with_timeit_module.py]

import timeit
import random

for i in range(10000,1000001,20000):
    t = timeit.Timer("random.randrange(%d) in x"%i, "from __main__
        import random,x")
    x = list(range(i))
    lst_time = t.timeit(number=1000)
    x = {j:None for j in range(i)}
    d_time = t.timeit(number=1000)
    print("%d,%10.3f,%10.3f" % (i, lst_time, d_time))
```

We clearly see that the membership operation for lists is $\mathcal{O}(n)$ and for dictionaries is $\mathcal{O}(1)$):

```
10000,     0.192,     0.002
30000,     0.600,     0.002
50000,     1.000,     0.002
```

```
70000,    1.348,    0.002
90000,    1.755,    0.002
110000,    2.194,    0.002
130000,    2.635,    0.002
150000,    2.951,    0.002
170000,    3.405,    0.002
190000,    3.743,    0.002
210000,    4.142,    0.002
230000,    4.577,    0.002
250000,    4.797,    0.002
270000,    5.371,    0.002
290000,    5.690,    0.002
310000,    5.977,    0.002
```

Iterating over Dictionaries

A loop over a dictionary iterates over its keys by default. The keys will appear in an arbitrary order but we can use `sorted()` to iterate over the items in a sorted way. This also works for the attributes `keys()`, `values()`, and `items()`:

```
>>> for key in sorted(dict.keys()):
...     print key, dict[key]
```

Generators can be used to create a list of key-items for a dictionary:

```
def items_in_key_order(d):
   for key in sorted(d):
      yield key, d[key]
```

3.3 Python's `collection` Data Types

Python's `collections` module implements specialized container data types providing high-performance alternatives to the general purpose built-in containers.

Default Dictionaries

`Default dictionaries` are an additional unordered mapping type provided by Python's `collections.defaultdict`. They have all the operators and methods that a built-in dictionary has, but they also handle missing keys:

```
[example_defaultdict.py]

from collections import defaultdict

def  defaultdict_example():
    pairs = {('a', 1), ('b',2), ('c',3)}
    d1 = {}
    for key, value in pairs:
        if key not in d1:
            d1[key] = []
        d1[key].append(value)
    print(d1)

    d2 = defaultdict(list)
    for key, value in pairs:
        d2[key].append(value)
    print(d2)
```

Ordered Dictionaries

Ordered dictionaries are an ordered mapping type provided by Python's `collections.OrderedDict`. They have all the methods and properties of a built-in `dict`, but in addition they store items in the *insertion order*:

```
[example_OrderedDict.py]

from collections import OrderedDict

   pairs = [('a', 1), ('b',2), ('c',3)]
    d1 = {}
    for key, value in pairs:
        if key not in d1:
            d1[key] = []
```

```
        d1[key].append(value)
    for key in d1:
        print(key, d1[key])

    d2 = OrderedDict(pairs)
    for key in d2:
        print(key, d2[key])
```

We can create ordered dictionaries incrementally:

```
>>> tasks = collections.OrderedDict()
>>> tasks[8031] = "Backup"
>>> tasks[4027] = "Scan Email"
>>> tasks[5733] = "Build System"
>>> tasks
OrderedDict([(8031, 'Backup'), (4027, 'Scan Email'), (5733, 'Build
    System')])
```

If we change a key value, the order is not changed. In addition, to move an item to the end we should delete and re-insert it (or call `popitem()` to remove and return the last key-value item).

In general, using an ordered dictionary to produce a sorted dictionary makes sense only if we *expect to iterate over the dictionary multiple times*, and if we *do not expect to do any insertions* (or very few).

Counter Dictionaries

A specialized `Counter` type (subclass for counting objects) is provided by Python's `collections.Counter`:

```
[example_Counter.py]

from collections import Counter

def  Counter_example():
    seq1 = [1, 2, 3, 5, 1, 2, 5, 5, 2, 5, 1, 4]
    seq_counts = Counter(seq1)
    print(seq_counts)
```

```
    ''' we can increment manually or use the update() method '''
    seq2 = [1, 2, 3]
    seq_counts.update(seq2)
    print(seq_counts)

    seq3 = [1, 4, 3]
    for key in seq3:
        seq_counts[key] += 1
    print(seq_counts)

    ''' we can use set operations such as a-b or a+b '''
    seq_counts_2 = Counter(seq3)
    print(seq_counts_2)
    print(seq_counts + seq_counts_2)
    print(seq_counts - seq_counts_2)
```

3.4 Further Examples

Counting Frequency of Items

In the example below we use `collections.Counter()`'s `most_common()` method to find the top N recurring words in a sequence:

```
[find_top_N_recurring_words.py]

from collections import Counter

def find_top_N_recurring_words(seq, N):
   dcounter = Counter()
    for word in seq.split():
        dcounter[word] += 1
    return dcounter.most_common(N)

def test_find_top_N_recurring_words(module_name='this module'):
    seq = 'buffy angel monster xander a willow gg buffy the monster
        super buffy angel'
    N = 3
    assert(find_top_N_recurring_words(seq, N) == [('buffy', 3),
        ('monster', 2), ('angel', 2)])
```

```
    s = 'Tests in {name} have {con}!'
    print(s.format(name=module_name, con='passed'))
```

Counting Frequency of Unique Words

The program below counts all the unique words in a file:

```
[count_unique_words.py]

import collections
import string
import sys

def count_unique_word():
    words = collections.defaultdict(int)
    strip = string.whitespace + string.punctuation + string.digits
        + "\"'"
    for filename in sys.argv[1:]:
        with open(filename) as file:
            for line in file:
                for word in line.lower().split():
                    word = word.strip(strip)
                    if len(word) > 2:
                        words[word] = +1
    for word in sorted(words):
        print("'{0}' occurs {1} times.".format(word, words[word]))
```

Anagrams

The following program finds whether two words are anagrams. Since sets do not count occurrence, and sorting a list is $\mathcal{O}(n \log n)$, hash tables can be a good solution for this type of problem:

```
[anagram.py]
```

```
def is_anagram(s1, s2):
    '''
    >>> is_anagram('cat', 'tac')
    True
    >>> is_anagram('cat', 'hat')
    False
    '''
    counter = Counter()
    for c in s1:
        counter[c] += 1
    for c in s2:
        counter[c] -= 1
    for i in counter.values():
        if i:
            return False
    return True
```

Sums of Paths

The following program uses two different dictionary containers to determine the number of ways two dices can sum to a certain value:

```
[find_dice_probabilities.py]

from collections import Counter, defaultdict

def find_dice_probabilities(S, n_faces=6):
    if S > 2*n_faces or S < 2:
      return None
    cdict = Counter()
    ddict = defaultdict(list)

    for dice1 in range(1, n_faces+1):
        for dice2 in range(1, n_faces+1):
            t = [dice1, dice2]
            cdict[dice1+dice2] += 1
            ddict[dice1+dice2].append( t)
```

```
    return [cdict[S], ddict[S]]

def test_find_dice_probabilities(module_name='this module'):
    n_faces = 6
    S = 5
    results = find_dice_probabilities(S, n_faces)
    print(results)
    assert(results[0] == len(results[1]))
```

Finding and Deleting Duplicates

The program below uses dictionaries to find and delete all the duplicate characters in a string:

```
[delete_duplicate_char_str.py]

from collections import Counter

def delete_unique(str1):
    '''
    >>> delete_unique("Trust no one")
    'on'
    >>> delete_unique("Mulder?")
    ''
    '''
    str_strip = ''.join(str1.split())
    repeat = Counter()
    for c in str_strip:
        repeat[c] += 1
    result = ''
    for c, count in repeat.items():
        if count > 1:
            result += c
    return result

def removing_duplicates_seq(str1):
    '''
    >>> delete_unique("Trust no one")
```

```
    'on'
    >>> delete_unique("Mulder?")
    ''
    '''
    seq = str1.split()
    result = set()
    for item in seq:
        if item not in result:
            #yield item
            result.add(item)
    return result
```

Chapter 4

Python's Structure and Modules

4.1 Control Flow

if

The `if` statement substitutes the `switch` or `case` statements in other languages:[1]

```
>>> x = int(input("Please enter a number: "))
>>> if x < 0:
...     x = 0
...     print "Negative changed to zero"
>>> elif x == 0:
...     print "Zero"
>>> elif x == 1:
...     print "Single"
>>> else:
...     print "More"
```

[1]Note that colons are used with `else`, `elif`, and in any other place where a suite is to follow.

for

Python's `for` statement iterates over the items of any sequence (*e.g.*, a list or a string), in the order that they appear in the sequence:

```
>>> a = ["buffy", "willow", "xander", "giles"]
>>> for i in range(len(a)):
...     print(a[i])
buffy
willow
xander
giles
```

False and True in Python

False is defined by either the predefined constant `False`, the object `None`, or by an empty sequence or collection (empty string `' '`, list `[]`, or tuple `()`). Anything else is `True`.

The Google Python Style guide sets the following rules for using implicit `False` in Python:

- ⋆ Never use `==` or `!=` to compare *singletons*, such as the built-in variable `None`. Use `is` or `is not` instead.
- ⋆ Beware of writing `if x` when you mean `if x is not None`.
- ⋆ Never compare a boolean variable to `False` using `==`. Use `if not x:` instead. If you need to distinguish `False` from `None` then chain the expressions, such as `if not x and x is not None`.
- ⋆ For sequences (strings, lists, tuples), use the fact that empty sequences are `False`, so `if not seq` or `if seq` is preferable to `if len(seq)` or `if not len(seq)`.
- ⋆ When handling integers, implicit `False` may involve more risk than benefit, such as accidentally handling `None` as 0:

```
[Good]
     if not users: print 'no users'
```

```
     if foo == 0: self.handle_zero()
     if i % 10 == 0: self.handle_multiple_of_ten()

[Bad]
     if len(users) == 0: print 'no users'
     if foo is not None and not foo: self.handle_zero()
     if not i % 10: self.handle_multiple_of_ten()
```

`yield` vs. `return`

One great feature in Python is how it handles iterability. An *iterator* is a container object that implements the iterator protocol and is based on two methods: `next`, which returns the next item in the container, and `__iter__` which returns the iterator itself.

In this context, `yield` comes in hand. The difference between `yield` and `return` is that the former returns each value to the caller and then *only returns to the caller when all values to return have been exhausted*, and the latter *causes the method to exit and return control to the caller*.

The `yield` paradigm becomes important in the context of *generators*, which are a powerful tool for creating iterators. Generators are like regular functions but instead of returning a final value in the end, they use the `yield`. Their values are extracted one at time by calling `__next__()` until a `StopIteration` is raised:

```
>>> def f(a):
...     while a:
...         yield a.pop()
```

Generators should be considered when dealing with functions that return sequences or create a loop. For example, the following program implements a Fibonacci sequence using the iterator paradigm:

```
def fib_generator():
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a+b

if __name__ == '__main__':
```

```
    fib = fib_generator()
    print(next(fib))
    print(next(fib))
    print(next(fib))
    print(next(fib))
```

`break` vs. `continue`

The command `break`, *breaks out of the smallest enclosing* `for` or `while` loop, and switches control to the statement following the innermost loop in which the break statement appears.

Loop statements might have an `else` clause which is executed when the loop terminates through exhaustion of the list (with `for`) or when the condition becomes false (with `while`), but not when the loop is terminated by a `break` statement.

The command `continue` *continues with the next iteration of the loop*, and switches control to the start of the loop.

4.2 Modules in Python

Modules are defined using the built-in name `def`. When `def` is executed, a *function object* is created together with its *object reference*. If we do not define a `return` value, Python automatically returns `None` (like in `C`, we call the function a *procedure* when it does not return a value).

Default Values in Modules

Whenever you create a module, remember that mutable objects should not be used as default values in the function or method definition:

```
[Good]
def foo(a, b=None):
   if b is None:
      b = []
[Bad]
   def foo(a, b=[]):
```

Activation Records

An *activation record* happens every time we invoke a method: information is put in the *stack* to support invocation. Activation records process in the following order:

1. the actual parameters of the method are pushed onto the stack,
2. the return address is pushed onto the stack,
3. the top-of-stack index is incremented by the total amount required by the local variables within the method,
4. a jump to the method.

The process of unwinding an activation record happens in the following order:

1. the top-of-stack index is decremented by the total amount of memory consumed,
2. the returned address is popped off the stack,
3. the top-of-stack index is decremented by the total amount of memory by the actual parameters.

Checking the Existence of a Module

To check the existence of a module, we casn use the flag `-c`:

```
$ python -c "import decimas"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named decimas
```

Packages

A *package* is a directory that contains a set of modules and a file called `__init__.py` :

```
>>> import foldername.filemodulename
```

This is required to make Python treat the directories as containing packages, preventing directories with a common name (such as "string") from hiding valid modules that occur later on the module search path:

In the simplest case, `__init__.py` can be an empty file. It can also execute initialization code for the package or set the `__all__` variable:

```
__all__ = ["file1", ...]
```

(with no `.py` in the end).

In this case, the statement:

```
from foldername import *
```

means importing every object in the module, except those whose names begin with `__`, or if the module has a global `__all__` variable, the list in it.

The `__name__` Variable

Whenever a module is *imported*, Python stores the module's name in the variable `__name__`.

If we run the `.py` file directly, Python sets `__name__` to `__main__`, and every instruction following the below the statement:

```
if __name__ == '__main__':
```

is executed.

Byte-coded Compiled Modules

Byte-compiled code, in form of `.pyc` files, is used by the compiler to *speed-up the start-up time* (load time).

When the Python interpreter is invoked with the `-O` flag, optimized code is generated and stored in `.pyo` files (for example, assert statements are removed). This also can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer it.

The `sys` Module

The variable `sys.path` is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the

environment variable `PYTHONPATH`, or from a built-in default. You can modify it using list operations:

```
>>> import sys
>>> sys.path.append( /buffy/lib/ python )
```

The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts. The variable `sys.argv` allows the recovering of the arguments passed in the command line:

```
import sys

''' print command line arguments '''
for arg in sys.argv[1:]:
    print arg
```

The built-in method `dir()` is used to find *which names a module defines* and includes all types of names: variables, modules, functions:

```
>>> import sys
>>> dir(sys)
[  __name__   ,   argv   ,  builtin_module_names  ,   copyright   ,
     exit   ,   maxint   ,   modules   ,   path   ,   ps1   ,
     ps2   ,   setprofile   ,   settrace   ,   stderr   ,
     stdin   ,   stdout   ,   version   ]
```

`dir()` is useful to find all the methods or attributes of an object.

4.3 File Handling

File handling is very easy in Python. For instance, the snippet below reads a file and remove any blank lines:

```
[remove_blank_lines.py]

import os
import sys

def read_data(filename):
```

```
    lines = []
    fh = None
    try:
        fh = open(filename)
        for line in fh:
            if line.strip():
                lines.append(line)
    except (IOError, OSError) as err:
        print(err)
    finally:
        if fh is not None:
            fh.close()
   return lines

def write_data(lines, filename):
    fh = None
    try:
        fh = open(filename, "w")
        for line in lines:
            fh.write(line)
    except (EnvironmentError) as err:
        print(err)
   finally:
        if fh is not None:
            fh.close()

def remove_blank_lines():
    if len(sys.argv) < 2:
        print ("Usage: noblank.py infile1 [infile2...]")
    for filename in sys.argv[1:]:
        lines = read_data(filename)
        if lines:
            write_data(lines, filename)
```

Methods for File Handling

The `open(filename, mode)` Method:

Returns a file object:

```
fin = open(filename, encoding="utf8")
fout = open(filename, "w", encoding="utf8")
```

The mode argument is optional and `read` will be assumed if it is omitted, the other options are:

- `r` for reading,
- `w` for writing (an existing file with the same name will be erased),
- `a` for appending (any data written to the file is automatically added to the end),
- and `r+` for both reading and writing.

The `read(size)` Method:

Reads some quantity from the data and returns it as a string. Size is an optional numeric argument and when it is omitted or negative, the entire contents of the file will be read and returned. If the end of the file has been reached, `read()` will return an empty string:

```
>>> f.read()
 'This is the entire file.\n'
>>> f.read()
 ' '
```

The `readline()` Method:

Reads a single line from the file. A newline character is left at the end of the string, and is only omitted on the last line of the file. This makes the return value unambiguous.

The `readlines()` Method:

Return a list containing all the lines of data in the file. If given an optional parameter `size`, it reads that many bytes from the file (and enough more to complete a line), and returns the lines from that. This is often used to allow efficient reading of a large file by lines, without having to load the entire file in memory:

```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the
   file\n']
```

The `write()` Method:

Writes the contents of a string to the file, returning `None`. Write bytes/bytearray object to the file if opened in binary mode or a string object if opened in text mode:

```
>>> f.write('This is a test\n')
```

The `tell()` and `seek()` Methods:

The `tell()` method returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file.

To change the file object's position, use `seek(offset, from-what)`. The position is computed from adding offset to a reference point and the reference point is selected by the from-what argument. A from-what value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point.

The `close()` Method:

Closes the file and free up any system resources taken up by the open file. It returns `True` if the file is closed.

The `input()` Method:

Accepts input from the user. This function takes an optional string argument (which will be printed in the console), then it will wait for the user to type in a response and to finish by pressing Enter (or Return).

If the user does not type any text but just presses Enter, the function returns an empty string; otherwise, it returns a string containing what the user typed, without any line terminator.

```
>>> def get_int(msg):
...     while True:
```

```
...     try:
...         i = int(input(msg))
...         return i
...     except ValueError as err:
...         print(err)
>>> age = get_int("Enter your age: ")
```

The `peek(n)` Method:

Returns `n` bytes without moving the file pointer position.

The `fileno()` Method:

Returns the underlying file's *file descriptor* (available only for file objects that have file descriptors).

The `shutil` Package

This package is useful for manipulating files in the system. For example, the following snippet gets a file and an extension from the command line and produces a copy of this file with its extension changed to the given string:

```
[change_ext_file.py]

import os
import sys
import shutil

def change_file_ext():
    if len(sys.argv) < 2:
        print("Usage: change_ext.py filename.old_ext 'new_ext'")
        sys.exit()
    name = os.path.splitext(sys.argv[1])[0] + "." + sys.argv[2]
    print (name)
    try:
        shutil.copyfile(sys.argv[1], name)
    except OSError as err:
        print (err)
```

The `pickle` Module

The `pickle` module can take almost any Python object (even some forms of Python code!), and convert it to a string representation. This process is called *pickling*. Reconstructing the object from the string representation is called *unpickling*.

If you have an object `x`, and a file object `f` that has been opened for writing, the simplest way to pickle the object takes only one line of code:

```
>>> pickle.dump(x, f)
```

Then, to unpickle this object:

```
>>> x = pickle.load(f)
```

Exporting Data with Pickle

The example below shows how to export serialized data with `pickle`:

```
[export_pickle.py]

import pickle

def export_pickle(data, filename='test.dat', compress=False):
    fh = None
    try:
        if compress:
            fh = gzip.open(filename, "wb") # write binary
        else:
            fh = open(filename, "wb") # compact binary pickle format
            pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)

    except(EnvironmentError, pickle.PickingError) as err:
        print("{0}: export error:
            {1}".format(os.path.basename(sys.argv[0], err)))
        return False

    finally:
```

```
        if fh is not None:
            fh.close()
```

Reading Data with Pickle

The example below shows how to read a pickled data:

```
[import.py]

import pickle

def import_pickle(filename):
    fh = None
    try:
        fh = open(filename, "rb")
        mydict2 = pickle.load(fh)
        return mydict2
    except (EnvironmentError) as err:
        print ("{0}: import error:
            {0}".format(os.path.basename(sys.arg[0]), err))
        return false
    finally:
        if fh is not None:
            fh.close()
```

The `struct` Module

We can convert Python objects to and from suitable binary representation using `struct`. This object can handle only strings with a specific length.

`struct` allows the creation of a function that takes a string and return a byte object with an integer length count and a sequence of `UTF-8` encoded bytes.

Some methods are: `struct.pack()` (takes a struct format string and values and returns a byte object), `struct.unpack()` (takes a format and a byte or bytearray object and returns a tuple of values), and `struct.calcsize()` (takes a format and returns how many bytes the struct occupies):

```
>>> data = struct.pack("<2h", 11, -9)
>>> items = struct.unpack("<2h", data) # little endian
```

4.4 Error Handling in Python

There are two distinguishable kinds of errors when we compile a program in Python: *syntax errors* (parsing errors) and *exceptions* (errors detected during execution, not unconditionally fatal). While syntax errors will never allow the program to be compiled, exceptions can only be detected in some cases and for this reason they should be carefully handled.

Handling Exceptions

When an exception is raised and not handled, Python outputs a *traceback* along with the exception's error message. A traceback is a list of all the calls made from the point where the unhandled exception occurred back to the top of the call *stack*.

We can handle predictable exceptions with the `try-except-finally` paradigm:

```
try:
    try_suite
except exception1 as variable1:
    exception_suite1
...
except exceptionN as variableN:
    exception_suiteN
```

If the statements in the try block's suite are all executed without raising an exception, the except blocks are skipped. If an exception is raised inside the try block, control is immediately passed to the suite corresponding to the first matching exception. This means that any statements in the suite that follow the one that caused the exception will not be executed:

```
while 1:
    try:
        x = int(raw_input("Please enter a number: "))
```

```
        break
    except ValueError:
        print "Oops! That was no valid number. Try again..."
```

The `raise` statement allows the programmer to force a specified exception to occur:

```
import string
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

We also can use else:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

The Google Python Style Guide for Exceptions

Exceptions must be used carefully and must follow certain conditions:

- ⋆ Raise exceptions like this: `raise MyException('Error message')` or `raise MyException`. Do not use the two-argument form.
- ⋆ Modules or packages should define their own domain-specific base ex-

ception class, which should be inherit from the built-in `Exception` class. The base exception for a module should be called `Error`.

```
class Error(Exception):
    pass
```

- ⋆ Never use catch-all except: statements, or catch `Exception` or `StandardError`, unless you are re-raising the exception or in the outermost block in your thread (and printing an error message).
- ⋆ Minimize the amount of code in a `try/except` block. The larger the body of the try, the more likely that an exception will be raised by a line of code that you didn't expect to raise an exception. In those cases, the try/except block hides a real error.
- ⋆ Use the `finally` clause to execute code whether or not an exception is raised in the try block. This is often useful for clean-up, *i.e.*, closing a file.
- ⋆ When capturing an exception, use `as` rather than a comma. For example:

```
try:
    raise Error
except Error as error:
    pass
```

4.5 Special Methods

The `range()` Method

This method generates lists containing arithmetic progressions. It is useful when you need to iterate over a sequence of numbers:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(4, 10)
[4, 5, 6, 7, 8, 9]
>>> range(0, 10, 3)
```

```
[0, 3, 6, 9]
```

The `enumerate()` Method

Returns a tuple with the item and the index values from an iterable. For example, we can use `enumerate` to write our own `grep` function, which gets a word and files from the command line and outputs where the word appears:

```
[grep_word_from_files.py]

import sys

def grep_word_from_files():
    word = sys.argv[1]
    for filename in sys.argv[2:]:
        with open(filename) as file:
            for lino, line in enumerate(file, start=1):
                if word in line:
                    print("{0}:{1}:{2:.40}".format(filename, lino,
                        line.rstrip()))

if __name__ == '__main__':
    if len(sys.argv) < 2:
        print("Usage: grep_word_from_files.py word infile1
            [infile2...]")
        sys.exit()
    else:
        grep_word_from_files()
```

The `zip()` Method

The `zip` function takes two or more sequences and creates a new sequence of tuples where each tuple contains one element from each list:

```
>>> a = [1, 2, 3, 4, 5]
>>> b = ['a', 'b', 'c', 'd', 'e']
>>> zip(a, b)
```

```
<zip object at 0xb72d65cc>
>>> list(zip(a,b))
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

The `filter(function, sequence)` Method

This method returns a sequence consisting of items for which function (item) is true:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
>>> f(33)
False
>>> f(17)
True
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

The `map(function, list)` Function

It applies a function to every item of an iterable and then returns a list of the results:

```
>>> def cube(x): return x*x*x
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
>>> seq = range(8)
>>> def square(x): return x*x
>>> map(None, seq, map(square, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7,
   49)]
```

The `lambda` Function

The `lambda` function is a dynamic way of compacting functions inside the code. For example, the function:

```
>>> def area(b, h):
...     return 0.5 * b * h
```

```
...
>>> area(5,4)
10.0
```

could be rather written as

```
>>> area = lambda b, h: 0.5 * b * h
>>> area(5, 4)
10.0
```

Chapter 5

Object-Oriented Design

Suppose we want to define an object in Python to represent a circle. We could remember about Python's `collections` module and create a **named tuple** for our circle:

```
>>> circle = collections.namedtuple("Circle", "x y radius")
>>> circle
<class '__main__.Circle'>
>>> circle = circle(13, 84, 9)
>>> circle
Circle(x=13, y=84, radius=9)
```

However, many things are missing here. First, there are no guarantees that anyone who uses our circle data is not going to type an invalid input value, such as a negative number for the radius. Second, how could we also associate to our circle some operations that are proper from it, such as its area or perimeter?

For the first problem, we can see that the inability to validate when creating an object is a really bad aspect of taking a purely *procedural* approach in programming. Even if we decide to include many exceptions handling the invalid inputs for our circles, we still would have a data container that is not intrinsically made and validated for its real purpose. Imagine now if we had chosen a **list** instead of the named tuple, how would we handle the fact that lists have sorting properties?

It is clear that we need to create an object that has **only** the proprieties that we expect it to have. In other words, we want to find a way to package data and restrict its methods. That is what *object-oriented programming*

allows you to do: to *create your own custom data type*.

5.1 Classes and Objects

Classes gather special predefined data and methods together. We use them by creating *instances*:

```
class ClassName:
    <statement-1>
    .g
    <statement-N>

>>> x = ClassName() # class instantiation
```

Class Instantiation

Class instantiation uses function notation to create objects in a known initial state. The instantiation operation creates an empty object which has individuality[1].

When an object is created in Python, first the special method `__new__()` is called (the *constructor*) and then `__init__()` initializes it.

Attributes

Objects have *attributes* (methods and data) from their Classes. Method attributes are functions whose first argument is the instance on which it is called to operate (conventionally called `self`).

References to names in modules are attribute references: in the expression modname.funcname, modname is a module object and funcname is one of its attribute. Attributes may be read-only or writable. Writable attributes may be deleted with the `del` statement.

[1]Multiple names (in multiple scopes) can be bound to the same object (also know as *aliasing*).

Namespaces

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries.

Examples of namespaces are: the set of built-in names, the global names in a module, and the local names in a function invocation. The statements executed by the top-level invocation of the interpreter, either reading from a script file or interactively, are considered part of a module called `__main__`.

Scope

A *scope* is a textual region of a Python program where a namespace is directly accessible. Although scopes are determined statically, they are used dynamically.

The global scope of a function defined in a module is the module's namespace. When a class definition is entered, a new namespace is created, and used as the local scope.

5.2 Principles of OOP

inheritance

Inheritance (or specialization) is the procedure of creating a new class that *inherits* all the attributes from the super class (also called *base class*).

If a class inherits from no other base classes, we should explicitly inherit it from Python's highest class, `object`:

```
class OuterClass(object):
      class InnerClass(object):
```

Inheritance is described as an **is-a** relationship.

Polymorphism

Functions in Python are virtual, so any method can be overridden in a subclass. *Polymorphism* is the principle where methods can be redefined inside subclasses.

If we need to recover the superclass's method, we call it using the built-in `super()`.

Aggregation

Aggregation (or composition) defines the process where a class includes one of more instance variables that are from other classes. It is a **has-a** relationship.

5.3 An Example of a Class

The example below illustrates how we could write a circle data container using the object-oriented design paradigm.

First, we create a class called `Point` with general data and methods attributes:

```
[ShapeClass.py]

import math

class Point:
   def __init__(self, x = 0, y = 0):
      self.x = x # data attribute
      self.y = y

   def distance_from_origin(self): # method attribute
      return math.hypot(self.x, self.y)

   def __eq__(self, other):
      return self.x == other.x and self.y == other.y

   def __repr__(self):
      return "point ({0.x!r}, {0.y!r})".format(self)

   def __str__(self):
      return "({0.x!r}, {0.y!r})".format(self)
```

Then we use *inheritance* to create a Circle subclass from it:

```
class Circle(Point):
```

```
  def __init__(self, radius, x=0, y=0):
     super().__init__(x,y) # creates/initializes
     self.radius = radius

  def edge_distance_from_origin(self):
     return abs(self.distance_from_origin() - self.radius)

  def area(self):
     return math.pi*(self.radius**2)

  def circumference(self):
     return 2*math.pi*self.radius

  def __eq__(self, other): # avoid infinite recursion
     return self.radius == other.radius and super().__eq__(other)

  def __repr__(self):
     return "circle ({0.radius!r}, {0.x!r})".format(self)

  def __str__(self):
     return repr(self)
```

Testing our class:

```
>>> import ShapeClass as shape
>>> a = shape.Point(3,4)
>>> a
point (3, 4)
>>> repr(a)
'point (3, 4)'
>>> str(a)
'(3, 4)'
>>> a.distance_from_origin()
5.0
>>> c = shape.Circle(3,2,1)
>>> c
circle (3, 2)
>>> repr(c)
'circle (3, 2)'
```

```
>>> str(c)
'circle (3, 2)'
>>> c.circumference()
18.84955592153876
>>> c. edge_distance_from_origin()
0.7639320225002102
```

5.4 Python Design Patterns

Design patterns are an attempt to bring a formal definition for correctly designed structures to software engineering. There are many different design patterns to solve distinct problems.

Decorator Pattern

Decorators (used with a @ notation) are a tool to elegantly specify some transformation on functions and methods. The decorator pattern allows us to *wrap* an object that provides core functionality with other objects that alter that functionality.

With decorators, the snippet below:

```
class NiceClass(object):
    def method(self):
      method = my_decorator(method)
```

can be written as:

```
class NiceClass(object):
    @my_decorator
    def method(self):
```

Let's look at a reals examples. For instance, a sum function that would be otherwise written as:

```
[example_decorators.py]
def sum(func):
    s = 0
    for i in func():
```

```
        s += i
    return s

def interate():
    a = []
    for i in range(10):
        a.append(i)
    return a

print sum(interate)
```

can be instead decorated:

```
@sum
def interate():
    a = []
    for i in range(10):
        a.append(i)
    return a

print interate
```

Another useful example is a logger decorator:

```
def logger(func):
    def inner(*args): #1
        print "Arguments were: {0}".format(args)
        return func(*args)
    return inner

@logger
def foo(x, y):
    return x+y

print foo(1, 2)
```

Another example is a custom benchmarking function:

```
[do_benchmark.py]
import random
```

```
def benchmark(func):
    import time
    def wrapper(*args, **kwargs):
        t = time.clock()
        res = func(*args, **kwargs)
        print("\t%s" % func.__name__, time.clock()-t)
        return res
    return wrapper

@benchmark
def random_tree(n):
    temp = [n for n in range(n)]
    for i in range(n+1):
        temp[random.choice(temp)] = random.choice(temp)
    return temp
```

The most common decorators are `@classmethod` and `@staticmethod`, for converting ordinary methods to class methods or static methods.

Observer Pattern

The *observer pattern* is useful when we want to have a core object that maintains certain values, and then having some observers to create serialized copies of that object.

This can be implemented by using the `@properties` decorator, placed before our functions. This will control attribute access, for example, to make an attribute to be read-only.

```
@property
def radius(self):
   return self.__radius
```

Singleton Pattern

A class follows the *singleton pattern* if it allows exactly one instance of a certain object to exist.

Since Python does not have private constructors, we use the `__new__` class method to ensure that only one instance is ever created. For example, in the

snippet below, the two objects are the same object (they are equal and are in the same address):

```
>>> class SinEx:
...     _sing = None
...     def __new__(self, *args, **kwargs):
...        if not self._sing:
...           self._sing = super(SinEx, self).__new__(self, *args,
    **kwargs)
...        return self._sing

>>> x = SinEx()
>>> x
<__main__.SinEx object at 0xb72d680c>
>>> y = SinEx()
>>> x == y
True
>>> y
<__main__.SinEx object at 0xb72d680c>
```

Chapter 6

Advanced Topics

6.1 Multiprocessing and Threading

Each program in the operational system is a separate *process*. Each process has one or more *threads*. If a process has several threads, they appear to run simultaneously.

In a program, two approaches can be used to spread the workload:

Multiple processes Multiple processes have separate regions of memory and can only communicate by special mechanisms. The processor loads and saves a separate set of registers for each thread, which inconvenient for communication and data sharing. In Python, multiple processes are handled by the `subprocess` module.

Multiple threads Multiple threads in a single process have access to the same memory. Threads share the process's resources, including the heap space. But each thread still has it own stack. In Python, threads are handled with the `threading` module.

Although Python has a threading mechanism, it does not support **true** parallel execution. However, it is possible is to use parallel processes, which in modern operating systems are really efficient.

The `subprocess` Module

Used to create a pair of "parent-child" programs. The parent program is started by the user and this in turn runs instances of the child program, each

with different work to do. Child processing allows to take advantage of multicore processor, leaving concurrency issues to be handled by the operational system.

The `threading` Module

Complexity arises when we want to separate threads. Since they share data, we need to be careful with a policy for locks.

To create multiple threads the `threading.thread()` method passes a callable object:

```
import threading

def worker(num):
    """thread worker function"""
    print 'Worker: %s' % num
    return

threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()
```

Additionally, we can use the `queue.queue()` class to handle all the locking internally. This will serialize accesses, meaning that only one thread at time has access to the data (FIFO).

Since the program will not terminate while there are running threads, the ones that are finished will appear to be still running. The solution is to transform threads into `daemons`. In this case, the program terminates as soon as there is no daemon threads running.

Mutexes and Semaphores

A *mutex* is like a lock. Mutexes are used in parallel programming to ensure that only one thread can access a shared resource at a time.

For example, say one thread is modifying an array. When it has gotten halfway through the array, the processor switches to another thread. If we

were not using mutexes, the thread could try to modify the array as well, at the same time.

Conceptually, a mutex is an integer that starts at 1. Whenever a thread needs to alter the array, it "locks" the mutex. This causes the thread to wait until the number is positive and then decreases it by one. When the thread is done modifying the array, it "unlocks" the mutex, causing the number to increase by 1.

Semaphores are more general than mutexes. A semaphore's integer may start at a number greater than 1. The number at which a semaphore starts is the number of threads that may access the resource at once. Semaphores support "wait" and "signal" operations, which are analogous to the "lock" and "unlock" operations of mutexes.

Deadlock and Spinlock

Deadlock is a problem that sometimes arises in parallel programming, when two threads are stuck indefinitely. We can prevent deadlock if we assign an order to our locks and require that locks will always be acquired in order (this is a very general and a not precise approach).

Spinlock is a form of *busy waiting*, that can be useful for high-performance computing (HPC) (when the entire system is dedicated to a single application and exactly one thread per core). It takes less time than a semaphore.

6.2 Good Practices

Virtual Environments

The more projects you have, the more likely it is that you will be working with different versions of Python itself, or at least different versions of Python libraries. For this reason, we use virtual environments (virtualenvs or venvs).

Virtualenv

To create a virtual environment:

```
$ virtualenv venv
```

To begin using the virtual environment, it needs to be activated:

```
$ source venv/bin/activate
```

If you are done working in the virtual environment for the moment, you can deactivate it:

```
$ deactivate
```

To delete a virtual environment, just delete its folder.

Virtualenvwrapper

Virtualenvwrapper provides some addtional commands and places all virtual environments in one place:

```
$ pip install virtualenvwrapper
```

Then edit your `.bashrc` file:

```
export WORKON_HOME=$HOME/.virtualenvs
export PROJECT_HOME=$HOME/Devel
source /usr/local/bin/virtualenvwrapper.sh
```

To create a virtual environment:

```
$ mkvirtualenv test
```

To check if it is working:

```
$ which python
```

To work on a virtual environment:

```
$ workon test
```

Deactivating is still the same:

```
$ deactivate
```

To delete it:

```
$ rmvirtualenv test
```

To check the configuration:

```
$ pip freeze
```

Other useful commands

```
lsvirtualenvList
cdvirtualenv
cdsitepackages
lssitepackages
```

Debugging

The Python debugger, pdb, can be found at http://pymotw.com/2/pdb/.

If you have some code in a source file and you want to debug it interactively, you can run Python with the `-i` switch:

```
$ python -i example.py
```

It also can be used in the command line:

```
$ python3 -m pdb program.py
```

or added as a module as the first statement of the function we want to examine:

```
import pdb
pdb.set_trace()
```

To perform the inspection, type: `s` for step, `p` for point, and `n` for next line, `list` to see the next 10 lines, and `help` for help.

Profiling

If a program runs very slowly or consumes far more memory than expected, the problem is most often due to our choice of algorithms or data structures or due to some inefficient implementation:

⋆ prefer tuples to list with read-only data;

- use *generators* rather than large lists or tuples to iteration;
- when creating large strings out of small strings, instead of concatenating the small, accumulate them all in a list and `join` the list of strings in the end.

The `cProfile` Package:

This package provides a detailed breakdown of call times and can be used to find performance bottlenecks.

```
import cProfile
cProfile.run('main()')
```

You can run it by typing:

```
$ python3 -m cProfile - o profile.day mymodule.py
$ python3 -m pstats
```

The `timeit` Package:

Used for timing small pieces of the code:

```
>>> import timeit
>>> timeit.timeit("x = 2 + 2")
0.034976959228515625
>>> timeit.timeit("x = sum(range(10))")
0.92387008666992188
> python -m timeit -s "import mymodule as m" "m.myfunction()"
```

6.3 Unit Testing

It is good practice to write tests for individual functions, classes, and methods, to ensure they behave to the expectations. Python's standard library provides two unit testing modules: `doctest` and `unittest`. There are also third-party testing tools: `nose` and `py.test`.

Test Nomenclature

Test fixtures: The code necessary to set up a test (for example, creating an input file for testing and deleting afterwards).

Test cases: The basic unit of testing.

Test suites: The collection of test cases, created by the subclass `unittest.testcase`, where each method has a name beginning with "test".

Test runner: An object that executes one of more test suites.

doctest

Good for tests inside the modules and functions' `docstrings`. To use it, we add the following lines in the module:

```
if __name__ = "__main__"
   import doctest
   doctest.testmod()
```

unittest

`unittest` is the standard testing package in Python. To use it, the test files should start with `test_`:

```
import unittest

class BasicsTestCase(unittest.TestCase):
    def test_find_name(self):
        self.assertTrue(1 == 1)
        self.assertFalse(1 == 2)

if __name__ == '__main__':
    unittest.main()
```

Which we run with:

```
$ python -m unittest <filename>
```

A fundamental concept here is of **test isolation**, where every test gets a new test object, failure does not stop tests, and test do not affect each other.

pytest

To install `pytest`, use:

```
$ pip install pytest
```

To use `pytest` we need to include a function that starts with `test` in a file that starts with `test`:

```
def func(x):
    return x + 1
def test_answer():
    assert func(3) == 51
```

Running:

```
$ python -m pytest
```

Part II

Algorithms are Fun!

Chapter 7

Asymptotic Analysis

Asymptotic analysis is a method to describe the limiting behavior and the performance of algorithms when applied to very large input datasets. To understand why asymptotic analysis is important, suppose you have to sort a billion of numbers ($n = 10^9$)[1] in some desktop computer. Suppose that this computer has a CPU clock time of 1 GHz, which roughly means that it executes 10^9 processor cycles (or operations) per second.[2] Then, for an algorithm that has a runtime of $\mathcal{O}(n^2)$, it would take approximately one billion of seconds to finish the sorting (in the worst case) which means one entire year!

In the Fig. 7 we can visualize how asymptotic analysis works, where several functions are plotted in terms of the number of inputs (n). We can see that the number of operations for any polynomial or exponential algorithm is unfeasible.

7.1 Complexity Classes

A *complexity class* is a set of problems with related complexity. A *reduction* is a transformation of one problem into another which is at least as difficult as the original one.

[1]Remember that for memory gigabytes means $10^9 \sim 1024^3 = 2^{30}$ bytes and for storage it means $1000^3 = 10^9$ bytes. Also, integers usually take 2 or 4 bytes each. However, for this example we are simplifying this by saying that a 'number' has 1 byte.

[2]In this exercise we are not considering other factors that would make the processing slower, such as RAM latency, copy cache operations, etc.

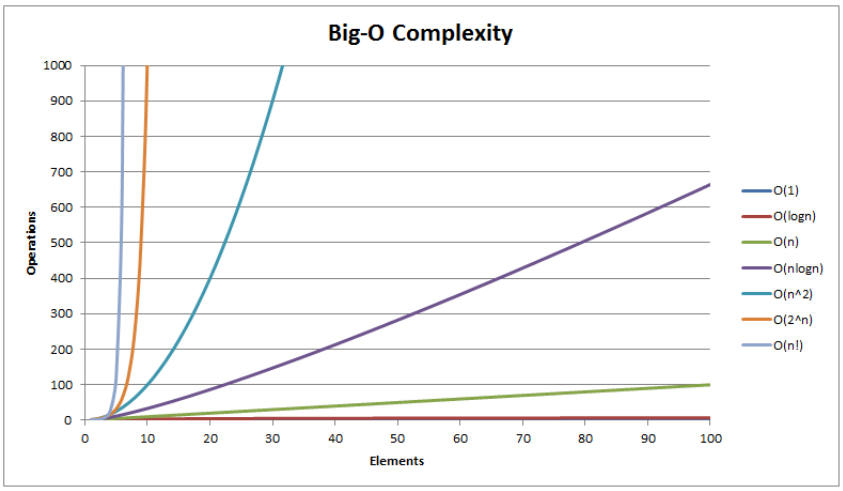


Figure 7.1: Asymptotic behaviour of many classes of functions.

P

P is the complexity class of *decision problems* that can be solved on a *deterministic Turing machine in polynomial time* (in the worst case). If we can turn a problem into a decision problem, the result would belong to **P**.

NP

NP is the complexity class of *decision problems* that can be solved on a *non-deterministic Turing machine (NTM) in polynomial time*. In other words, it includes all decision problems whose *yes instances* can be solved in polynomial time with the NTM.

A problem is called *complete* if all problems in the class are reduced to it. The subclass called *NP-complete* (**NPC**) contains the hardest problems in all of **NP**.

Any problem that is at least as hard (determined by polynomial-time reduction) as any problem in **NP**, but that need not itself be in **NP**, is called **NP-hard**. For example, finding the shortest route through a graph, which is called the *Travelling Salesman (or Salesrep) problem* (TSP).

P=NP?

The class **co-NP** is the class of the *complements* of **NP** problems. For every "yes" answer, we have the "no", and vice versa. If **NP** is truly asymmetric, then these two classes are different. Although, there is overlap between them since all of **P** lies in their intersection. Moreover, both the yes and no instances in **P** can be solved in polynomial time with an NTM.

What would happen if a **NPC** was found in a intersection of **N** and **co-NP**? First, it would mean that all of **NP** would be inside **co-NP**, so we would show **NP** = **co-NP** and the asymmetry would disappear. Second, since all of **P** is in this intersection, **P** = **NP**. If **P** = **NP**, we could solve any (decision) problem that had a practical (verifiable) solution.

However, it is (strongly) believed that **NP** and **co-NP** are different. For instance, no polynomial solution to the problem of factoring numbers was found, and this problem is in both **NP** and **co-NP**.

7.2 Recursion

In functions, the *three laws of recursion* are:

1. A recursive algorithm must have a *base case*.

2. A recursive algorithm must change its state and move toward the base case.

3. A recursive algorithm must call itself, recursively.

For every recursive call, the recursive function has to allocate memory on the *stack* for arguments, return address, and local variables, costing time to push and pop these data onto the stack. Recursive algorithms take at least $\mathcal{O}(n)$ space where n is the depth of the recursive call.

Recursion is very costly when there are duplicated calculations and/or there are overlap among subproblems. In some cases this can cause the stack to overflow. For this reason, where subproblems overlap, iterative solutions might be a better approach. For example, in the case of the Fibonacci series, the iterative solution runs on $\mathcal{O}(n)$ while the recursive solution runs on exponential runtime.

Recursive Relations

To describe the running time of recursive functions, we use recursive relations:

$$T(n) = a \cdot T(g(n)) + f(n),$$

where a represents the number of recursive calls, $g(n)$ is the size of each subproblem to be solved recursively, and $f(n)$ is any extra work done in the function. The following table shows examples of recursive relations:

$T(n) = T(n-1) + 1$	$\mathcal{O}(n)$	Processing a sequence
$T(n) = T(n-1) + n$	$\mathcal{O}(n^2)$	Handshake problem
$T(n) = 2T(n-1) + 1$	$\mathcal{O}(2^n)$	Towers of Hanoi
$T(n) = T(n/2) + 1$	$\mathcal{O}(\ln n)$	Binary search
$T(n) = T(n/2) + n$	$\mathcal{O}(n)$	Randomized select
$T(n) = 2T(n/2) + 1$	$\mathcal{O}(n)$	Tree transversal
$T(n) = 2T(n/2) + n$	$\mathcal{O}(n \ln n)$	Sort by divide and conquer

Divide and Conquer Algorithms

Recurrences for the *divide and conquer algorithms* have the form:

$$T(n) = a \cdot T(n/b) + f(n),$$

where we have a recursive calls, each with a percentage $1/b$ of the dataset. Summing to this, the algorithm does $f(n)$ of work. To reach the problem of T(1) = 1 in the final instance (leaf, as we will learn when we study trees), the *height* is defined as $h = \ln_b n$, Fig. 7.2.

7.3 Runtime in Functions

If an algorithm does not have any recursive calling, we only need to analyze its data structures and flow blocks. In this case, complexities of code blocks executed one after the other are just added and complexities of nested loops are multiplied.

If an algorithm has recursive calls, we can use the recursive functions defined in the previous section to find the runtime. When we write a recurrence relation for a function, we must write two equations, one for the general case and one for the base case (that should be $\mathcal{O}(1)$, so that $T(1) = 1$).

For instance, the algorithm to find the n^{th} element in a Fibonacci sequence (which is known as to be exponential):

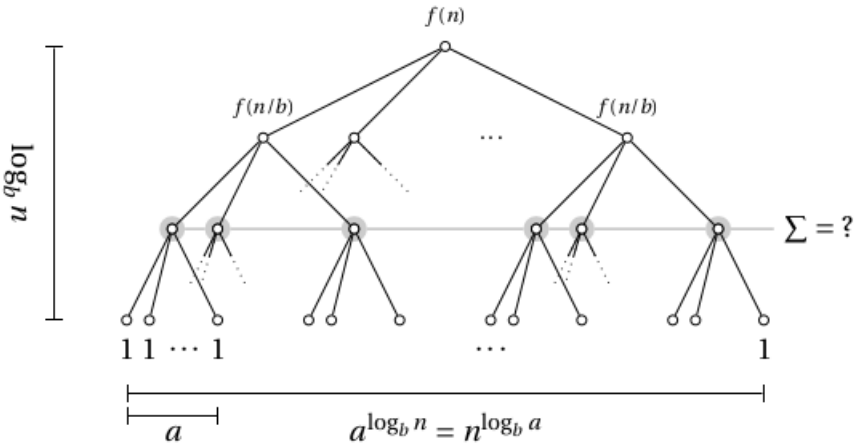


Figure 7.2: Tree illustrating divide and conquer recurrences.

```
[find_fibonacci_seq.py]

def find_fibonacci_seq_rec(n):
   if n < 2: return n
      return find_fibonacci_seq_rec(n - 1) +
         find_fibonacci_seq_rec(n - 2)
```

There, $g(n)$s are $n-2$ and $n-1$, a is 2, and $f(n)$ is 1, so the recursive relation is

$$T(n) = 2T(n-1) + 1.$$

Let us open this equation for each next recursion:

$$T(n) = 2^2 T(n-2) + 2 \to 2^k T(n-k) + k...$$

We need to make sure that the function have $\mathcal{O}(1)$ in the base case, where it is $T(1) = 1$, this means that $n - k = 1$ or $k = n - 1$. Plugging it back into the equation gives:

$$T(n) = 2^{n-1} + n - 1 \sim 2^n. \tag{7.3.1}$$

This proves that **this algorithm is exponential**! The same process can be done for each recursive relation.

In conclusion, the following table shows the runtime results for several known algorithms:

$\mathcal{O}(n^2)$	quadratic	insertion, selection sort
$\mathcal{O}(n \ln n)$	loglinear	algorithms breaking problem into smaller chunks per invocation, and then sticking the results together, quick and merge sort
$\mathcal{O}(n)$	linear	iteration over a list
$\mathcal{O}(\ln n)$	log	algorithms breaking problem into smaller chunks per invocation, searching a binary search tree
$\mathcal{O}(1)$	constant	hash table lookup/modification
$\mathcal{O}(n^k)$	polynomial	k-nested for loops over n
$\mathcal{O}(k^n)$	exponential	producing every subset of n items
$\mathcal{O}(n!)$	factorial	producing every ordering of n values

Chapter 8

Abstract Data Structures

An *abstract data type* (ADT) is a mathematical model for a certain class of data structures that have similar behavior. Different classes of abstract data types have many different but functionally equivalent data structures that implement them.

Data structures can be classified as either *contiguous* (composed of single slabs of memory) or *linked* (distinct chunks of memory bound together by pointers). In Python, contiguously-allocated structures include strings, lists, tuples, and dictionaries.

In the following sections we will learn how to write abstract data structures, whose can either use Python's builtin contiguous structures or be made of pointers.

8.1 Stacks

A *stack* is a linear data structure that can be accessed only at one of its ends (which we refers as the top) for either storing or retrieving. You can think of a stack as a huge pile of books on your desk.

Array access of elements in a stack is restricted since a stack is a *last-in-first-out* (LIFO) structure. However, stacks have the following operations running at $\mathcal{O}(1)$:

`push`: Insert an item at the top of the stack.

`pop`: Remove an item from the top of the stack.

`top/peek:` Look up the element on the top.

`empty/size:` Check whether the stack is empty or return its size.

Stacks in Python can be implemented with lists and the methods `append()` and `pop()`:

```
[stack.py]

class Stack(object):
    def __init__(self):
        self.content = []
        self.min_array = []
        self.min = float('inf')

    def push(self, value):
        if value < self.min:
            self.min = value
        self.content.append(value)
        self.min_array.append(self.min)

    def pop(self):
        if self.content:
            value = self.content.pop()
            self.min_array.pop()
            if self.min_array:
                self.min = self.min_array[-1]
            return value
        else:
            return 'Empty List. '

    def find_min(self):
        if self.min_array:
            return self.min_array[-1]
        else:
            return 'No min value for empty list.'

    def size(self):
        return len(self.content)

    def isEmpty(self):
```

```
        return not bool(self.content)

    def peek(self):
        if self.content:
            return self.content[-1]
        else:
            print('Stack is empty.')

    def __repr__(self):
        return '{}'.format(self.content)

if __name__ == '__main__':
    q = Stack()
    for i in range(15,20):
        q.push(i)
    for i in range(10,5,-1):
        q.push(i)
    for i in range(1, 13):
        print q.pop(), q.find_min()
```

Another approach to implement a stack is by thinking of it as a container of *nodes* (objects) following a LIFO order:

```
[linked_stack.py]

class Node(object):
    def __init__(self, value=None, pointer=None):
        self.value = value
        self.pointer = pointer

class Stack(object):
    def __init__(self):
        self.head = None

    def isEmpty(self):
        return not bool(self.head)

    def push(self, item):
        self.head = Node(item, self.head)
```

```
    def pop(self):
        if self.head:
            node = self.head
            self.head = node.pointer
            return node.value
        else:
            print('Stack is empty.')

    def peek(self):
        if self.head:
            return self.head.value
        else:
            print('Stack is empty.')

    def size(self):
        node = self.head
        count = 0
        while node:
            count +=1
            node = node.pointer
        return count

    def _printList(self):
        node = self.head
        while node:
            print(node.value)
            node = node.pointer

if __name__ == '__main__':
    stack = Stack()
    print("Is the stack empty? ", stack.isEmpty())
    print("Adding 0 to 10 in the stack...")
    for i in range(10):
        stack.push(i)
    stack._printList()
    print("Stack size: ", stack.size())
    print("Stack peek : ", stack.peek())
    print("Pop...", stack.pop())
    print("Stack peek: ", stack.peek())
    print("Is the stack empty? ", stack.isEmpty())
```

```
    stack._printList()
```

Stacks are suitable for *depth-first traversal* algorithms in *graphs*, as we will see in future chapters.

8.2 Queues

A *queue*, differently of a stack, is a structure where the first enqueued element (at the back) will be the first one to be dequeued (when it is at the front), being defined as a *first-in-first-out* (FIFO) structure. You can think of a queue as a line of people waiting for a roller-coaster ride.

Array access of elements in queues is restricted and queues should have the following operations running at $\mathcal{O}(1)$:

`enqueue:` Insert an item at the back of the queue.

`dequeue:` Remove an item from the front of the queue.

`peek/front:` Retrieve an item at the front of the queue without removing it.

`empty/size:` Check whether the queue is empty or give its size.

Again, we can write a queue class with Python's lists. However, this will be very inefficient (because it uses the method insert()):

```
[queue_inefficient.py]

class Queue(object):
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return not bool(self.items)

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
```

```
        return self.items.pop()

    def size(self):
        return len(self.items)

    def peek(self):
        return self.items[-1]

    def __repr__(self):
        return '{}'.format(self.items)

if __name__ == '__main__':
    queue = Queue()
    print("Is the queue empty? ", queue.isEmpty())
    print("Adding 0 to 10 in the queue...")
    for i in range(10):
        queue.enqueue(i)
    print("Queue size: ", queue.size())
    print("Queue peek : ", queue.peek())
    print("Dequeue...", queue.dequeue())
    print("Queue peek: ", queue.peek())
    print("Is the queue empty? ", queue.isEmpty())
    print(queue)
```

A better way would write a queue using two stacks (two lists) instead of one:

```
[queue.py]

class Queue(object):
    def __init__(self):
        self.in_stack = []
        self.out_stack = []

    def _transfer(self):
        while self.in_stack:
            self.out_stack.append(self.in_stack.pop())

    def enqueue(self, item):
        return self.in_stack.append(item)
```

```
    def dequeue(self):
        if not self.out_stack:
            self._transfer()
        if self.out_stack:
            return self.out_stack.pop()
        else:
            return "Queue empty!"

    def size(self):
        return len(self.in_stack) + len(self.out_stack)

    def peek(self):
        if not self.out_stack:
            self._transfer()
        if self.out_stack:
            return self.out_stack[-1]
        else:
            return "Queue empty!"

    def __repr__(self):
        if not self.out_stack:
            self._transfer()
        if self.out_stack:
            return '{}'.format(self.out_stack)
        else:
            return "Queue empty!"

    def isEmpty(self):
        return not (bool(self.in_stack) or bool(self.out_stack))

if __name__ == '__main__':
    queue = Queue()
    print("Is the queue empty? ", queue.isEmpty())
    print("Adding 0 to 10 in the queue...")
    for i in range(10):
        queue.enqueue(i)
    print("Queue size: ", queue.size())
    print("Queue peek : ", queue.peek())
    print("Dequeue...", queue.dequeue())
```

```
    print("Queue peek: ", queue.peek())
    print("Is the queue empty? ", queue.isEmpty())

    print("Printing the queue...")
    print(queue)
```

Another approach is to implement a queue as a container for nodes, as we have done for stacks, but now the nodes are inserted and removed in a FIFO order:

```
[linked_queue.py]

class Node(object):
    def __init__(self, value=None, pointer=None):
        self.value = value
        self.pointer = None

class LinkedQueue(object):
    def __init__(self):
        self.head = None
        self.tail = None

    def isEmpty(self):
        return not bool(self.head)

    def dequeue(self):
        if self.head:
            value = self.head.value
            self.head = self.head.pointer
            return value
        else:
            print('Queue is empty, cannot dequeue.')

    def enqueue(self, value):
        node = Node(value)
        if not self.head:
            self.head = node
            self.tail = node
        else:
            if self.tail:
```

```
                self.tail.pointer = node
            self.tail = node

    def size(self):
        node = self.head
        num_nodes = 0
        while node:
                num_nodes += 1
                node = node.pointer
        return num_nodes

    def peek(self):
        return self.head.value

    def _print(self):
        node = self.head
        while node:
            print(node.value)
            node = node.pointer

if __name__ == '__main__':
    queue = LinkedQueue()
    print("Is the queue empty? ", queue.isEmpty())
    print("Adding 0 to 10 in the queue...")
    for i in range(10):
        queue.enqueue(i)
    print("Is the queue empty? ", queue.isEmpty())
    queue._print()
    print("Queue size: ", queue.size())
    print("Queue peek : ", queue.peek())
    print("Dequeue...", queue.dequeue())
    print("Queue peek: ", queue.peek())
    queue._print()
```

Queues are necessary for *breath-first traversal* algorithms for graphs, as we will see in future chapters.

Deques

A *deque* is a double-ended queue, which can roughly be seen as an union of a stack and a queue.

Python has an efficient deque implementation, with fast appending and popping from both ends:

```
>>> from collections import deque
>>> q = deque(["buffy", "xander", "willow"])
>>> q
deque(['buffy', 'xander', 'willow'])
>>> q.append("giles")
>>> q
deque(['buffy', 'xander', 'willow', 'giles'])
>>> q.popleft()
'buffy'
>>> q.pop()
'giles'
>>> q
deque(['xander', 'willow'])
>>> q.appendleft('angel')
>>> q
deque(['angel', 'xander', 'willow'])
```

Interestingly, deques in Python are based on a *doubly linked list*,[1] not in dynamic arrays. It means that operations such as inserting an item anywhere are fast ($\mathcal{O}(1)$), but arbitrary index accessing are slow ($\mathcal{O}(n)$).

8.3 Priority Queues and Heaps

A *priority queue* is an abstract data type which is similar to a regular queue or stack, but where each element has a *priority* associated with it. If two elements have the same priority, they are served according to their order in the queue. A sensible implementation of a priority queue is given by a *heap* data structure.

[1] Linked lists are another abstract data structure that we will learn about at the end of this chapter. Doubly here means that their nodes have links to the next and to the previous node.

Heaps

Conceptually, a heap is a *binary tree* where each node is smaller (larger) than its children. We will learn about trees in the next chapters but we should already keep in mind that when modifications are made in a *balanced tree*, we can repair its structure with $\mathcal{O}(logn)$ runtimes.

Heaps are generally useful for applications that repeatedly access the smallest (largest) element in the list. A min-(max-)heap lets you to find the smallest (largest) element in $\mathcal{O}(1)$ and to extract/add/replace it in $\mathcal{O}(\ln n)$.

Python's `heapq` Package

Python implements heaps with the `heapq` module, which provides functions to insert and remove items while keeping the sequence as a heap.

We can use the `heapq.heapify(x)` method to transform a list into a heap, in-place, and in $\mathcal{O}(n)$ time:

```
>>> l = [4, 6, 8, 1]
>>> heapq.heapify(l)
>>> l
[1, 4, 8, 6]
```

Once we have a heap, the `heapq.heappush(heap, item)` method is used to push the item onto it:

```
>>> import heapq
>>> h = []
>>> heapq.heappush(h, (1, 'food'))
>>> heapq.heappush(h, (2, 'have fun'))
>>> heapq.heappush(h, (3, 'work'))
>>> heapq.heappush(h, (4, 'study'))
>>> h
[(1, 'food'), (2, 'have fun'), (3, 'work'), (4, 'study')]
```

The method `heapq.heappop(heap)` is used to pop and return the smallest item from the heap:

```
>>> l
[1, 4, 8, 6]
>>> heapq.heappop(l)
1
```

```
>>> l
[4, 6, 8]
```

The method `heapq.heappushpop(heap, item)` is used to push the item on the heap, then it pops and returns the smallest item from the heap. In a similar way, `heapq.heapreplace(heap, item)` will pop and return the smallest item from the heap, and then push the new item. In any case, the heap size does not change.

Several other operations are available with a heap proprieties. For example `heapq.merge(*iterables)` will merge multiple sorted inputs into a single sorted output (returning a iterator):

```
>>> for x in heapq.merge([1,3,5],[2,4,6]):
...     print(x,end="\n")
...
1
2
3
4
5
6
```

Finally, the methods `heapq.nlargest(n, iterable[, key])` and `heapq.nsmallest(n, iterable[, key])` will return a list with the n largest and smallest elements from the dataset defined by iterable.

A Class for a Heap

The following code does that, implementing a max-heap class for a given list:

```
[heapify.py]

class Heapify(object):
    def __init__(self, data=None):
        self.data = data or []
        for i in range(len(data)//2, -1, -1):
            self.__max_heapify__(i)

    def __repr__(self):
```

```
        return '{}'.format(self.data)

    def parent(self, i):
        return i >> 1

    def left_child(self, i):
        return (i << 1) + 1

    def right_child(self, i):
        return (i << 1) + 2 # +2 instead of +1 because it's
            0-indexed.

    def __max_heapify__(self, i):
        largest = i
        left = self.left_child(i)
        right = self.right_child(i)
        n = len(self.data)
        largest = (left < n and self.data[left] > self.data[i]) and
            left or i
        largest = (right < n and self.data[right] >
            self.data[largest]) and right or largest
        if i is not largest:
            self.data[i], self.data[largest] = self.data[largest],
                self.data[i]
            self.__max_heapify__(largest)

    def extract_max(self):
        n = len(self.data)
        max_element = self.data[0]
        self.data[0] = self.data[n - 1]
        self.data = self.data[:n - 1]
        self.__max_heapify__(0)
        return max_element

def test_Heapify():
    l1 = [3, 2, 5, 1, 7, 8, 2]
    h = Heapify(l1)
    assert(h.extract_max() == 8)
```

A Class for a Priority Queue

To conclude this section, the following example shows how to use the `heapq` package to implement a priority queue class:

```
[PriorityQueueClass.py]

import heapq

class PriorityQueue(object):
    def __init__(self):
        self._queue = []
        self._index = 0 # comparying same priority level

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]

class Item:
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return "Item({!r})".format(self.name)

def test_PriorityQueue():
    ''' push and pop are all O(logN) '''
    q = PriorityQueue()
    q.push(Item('test1'), 1)
    q.push(Item('test2'), 4)
    q.push(Item('test3'), 3)
    assert(str(q.pop()) == "Item('test2')")
```

8.4 Linked Lists

A linked list is a linear list of nodes, in general containing a value and a pointer (a reference) to the next node (and sometimes to the previous node). The last node in a linked list has a `None` reference.

An example of a node class is the following:

```
[node.py]

class Node(object):
    def __init__(self, value=None, pointer=None):
        self.value = value
        self.pointer = pointer

    def getData(self):
        return self.value

    def getNext(self):
        return self.pointer

    def setData(self, newdata):
        self.value = newdata

    def setNext(self, newpointer):
        self.pointer = newpointer

if __name__ == '__main__':
    L = Node("a", Node("b", Node("c", Node("d"))))
    assert(L.pointer.pointer.value=='c')
    print(L.getData())
    print(L.getNext().getData())
    L.setData('aa')
    L.setNext(Node('e'))
    print(L.getData())
    print(L.getNext().getData())
```

We can write a LIFO linked list as a collection of these nodes:

```
[likedlist_fifo.py]

from node import Node
```

```
class LinkedListLIFO(object):
    def __init__(self):
        self.head = None
        self.length = 0

    # print each node's value, starting from the head
    def _printList(self):
        node = self.head
        while node:
            print(node.value)
            node = node.pointer

    # delete a node, given the previous node
    def _delete(self, prev, node):
        self.length -= 1
        if not prev:
            self.head = node.pointer
        else:
            prev.pointer = node.pointer

    # add a new node, pointing to the previous node
    # in the head
    def _add(self, value):
        self.length += 1
        self.head = Node(value, self.head)

    # locate node with some index
    def _find(self, index):
        prev = None
        node = self.head
        i = 0
        while node and i < index:
            prev = node
            node = node.pointer
            i += 1
        return node, prev, i

    # locate node by value
    def _find_by_value(self, value):
```

```
        prev = None
        node = self.head
        found = 0
        while node and not found:
            if node.value == value:
                found = True
            else:
                prev = node
                node = node.pointer
        return node, prev, found

    # find and delete a node by index
    def deleteNode(self, index):
        node, prev, i = self._find(index)
        if index == i:
            self._delete(prev, node)
        else:
            print('Node with index {} not found'.format(index))

    # find and delete a node by value
    def deleteNodeByValue(self, value):
        node, prev, found = self._find_by_value(value)
        if found:
            self._delete(prev, node)
        else:
            print('Node with value {} not found'.format(value))

if __name__ == '__main__':
    ll = LinkedListLIFO()
    for i in range(1, 5):
        ll._add(i)
    print('The list is:')
    ll._printList()
    print('The list after deleting node with index 2:')
    ll.deleteNode(2)
    ll._printList()
    print('The list after deleting node with value 3:')
    ll.deleteNodeByValue(2)
    ll._printList()
    print('The list after adding node with value 15')
```

```
    ll._add(15)
    ll._printList()
    print("The list after deleting everything...")
    for i in range(ll.length-1, -1, -1):
        ll.deleteNode(i)
    ll._printList()
```

We can also write a class for a FIFO linked list:

```
[likedlist_fifo.py]

from node import Node

class LinkedListFIFO(object):
    def __init__(self):
        self.head = None
        self.length = 0
        self.tail = None # this is different from ll lifo

    # print each node's value, starting from the head
    def _printList(self):
        node = self.head
        while node:
            print(node.value)
            node = node.pointer

    # add a node in the first position
    # read will never be changed again while not empty
    def _addFirst(self, value):
        self.length = 1
        node = Node(value)
        self.head = node
        self.tail = node

    # delete a node in the first position, ie
    # when there is no previous node
    def _deleteFirst(self):
        self.length = 0
        self.head = None
        self.tail = None
```

```
        print('The list is empty.')

    # add a node in a position different from head,
    # ie, in the end of the list
    def _add(self, value):
        self.length += 1
        node = Node(value)
        if self.tail:
            self.tail.pointer = node
        self.tail = node

    # add nodes in general
    def addNode(self, value):
        if not self.head:
            self._addFirst(value)
        else:
            self._add(value)

    # locate node with some index
    def _find(self, index):
        prev = None
        node = self.head
        i = 0
        while node and i < index:
            prev = node
            node = node.pointer
            i += 1
        return node, prev, i

    # delete nodes in general
    def deleteNode(self, index):
        if not self.head or not self.head.pointer:
            self._deleteFirst()
        else:
            node, prev, i = self._find(index)
            if i == index and node:
                self.length -= 1
                if i == 0 or not prev :
                    self.head = node.pointer
                else:
```

```
                    prev.pointer = node.pointer
                if not self.tail == node:
                    self.tail = prev
            else:
                print('Node with index {} not found'.format(index))

if __name__ == '__main__':
    ll = LinkedListFIFO()
    for i in range(1, 5):
        ll.addNode(i)
    print('The list is:')
    ll._printList()
    print('The list after deleting node with index 2:')
    ll.deleteNode(2)
    ll._printList()
    print('The list after adding node with value 15')
    ll._add(15)
    ll._printList()
    print("The list after deleting everything...")
    for i in range(ll.length-1, -1, -1):
        ll.deleteNode(i)
    ll._printList()
```

Linked lists have a dynamic size at runtime and they are good for when you have an unknown number of items to store.

Insertion is $\mathcal{O}(1)$ but deletion and searching can be $\mathcal{O}(n)$ because locating an element in a linked list is slow and is it done by a *sequential search*. A good trick to delete a i node at $\mathcal{O}(1)$ is copying the data from $i+1$ to i and then to deleting the $i+1$ node.

Traversing a linked list backward or sorting it are both $\mathcal{O}(n^2)$.

8.5 Hash Tables

A hash table is used to associate keys to values, so that each key is associated with one or zero values. Insertion, removal, and lookup take $\mathcal{O}(1)$ time, provided that the hash is random.

We can build a class for a hash table by implementing an array of hash

buckets. For example, if a hash value is 42, and there are 5 buckets, the **mod** function is used to decide that the destination bucket is 2 (42 mod 5).

When two objects have the same hash value, we have a **collision**. One way to deal with this is to have a linked list for each bucket. In the worst-case scenario, each key hashes to the same bucket, so insertion, removal, and lookup will take $\mathcal{O}$(n) time.

```
[Hash_Table.py]

from linked_list_fifo import LinkedListFIFO

class HashTable(object):
    def __init__(self, slots=10):
        self.slots = slots
        self.table = []
        self.create_table()

    def hash_key(self, value):
        return hash(value)%self.slots

    def create_table(self):
        for i in range(self.slots):
            self.table.append([])

    def add_item(self, value):
        key = self.hash_key(value)
        self.table[key].append(value)

    def print_table(self):
        for key in range(len(self.table)):
            print "Key is %s, value is %s." %(key, self.table[key])

    def find_item(self, item):
        pos = self.hash_key(item)
        if item in self.table[pos]:
            return True
        else:
            return False

if __name__ == '__main__':
```

```
dic = HashTable(5)
for i in range(1, 40, 2):
    dic.add_item(i)
dic.print_table()
assert(dic.find_item(20) == False)
assert(dic.find_item(21) == True)
```

8.6 Additional Exercises

Stacks

Balancing Parenthesis in a String

The following example uses a stack to balance parenthesis in a string:

```
[balance_parenthesis_str.py]

from stack import Stack

def balance_par_str_with_stack(str1):
    s = Stack()
    balanced = True
    index = 0
    while index < len(str1) and balanced:
        symbol = str1[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()
        index += 1
    if balanced and s.isEmpty():
        return True
    else:
        return False

if __name__ == '__main__':
    print(balance_par_str_with_stack('((()))'))
    print(balance_par_str_with_stack('(()'))
```

Decimal to Binary

The following example uses a stack to transform a decimal number to binary number:

```
[dec2bin_with_stack.py]

from stack import Stack

def dec2bin_with_stack(decnum):
    s = Stack()
    str_aux = ''
    while decnum > 0:
        dig = decnum % 2
        decnum = decnum//2
        s.push(dig)
    while not s.isEmpty():
        str_aux += str(s.pop())
    return str_aux

if __name__ == '__main__':
    decnum = 9
    assert(dec2bin_with_stack(decnum) == '1001')
```

Stack with a Constant Lookup

The following example implements a stack that has $\mathcal{O}(1)$ minimum lookup:

```
[stack_with_min.py]

from stack import Stack

class NodeWithMin(object):
    def __init__(self, value=None, minimum=None):
        self.value = value
        self.minimum = minimum

class StackMin(Stack):
    def __init__(self):
        self.items = []
        self.minimum = None

    def push(self, value):
```

```
        if self.isEmpty() or self.minimum > value:
           self.minimum = value
        self.items.append(NodeWithMin(value, self.minimum))

    def peek(self):
        return self.items[-1].value

    def peekMinimum(self):
        return self.items[-1].minimum

    def pop(self):
        item = self.items.pop()
        if item:
            if item.value == self.minimum:
                self.minimum = self.peekMinimum()
            return item.value
        else:
            print("Stack is empty.")

    def __repr__(self):
        aux = []
        for i in self.items:
            aux.append(i.value)
        return '{}'.format(aux)

if __name__ == '__main__':
    stack = StackMin()
    print("Is the stack empty? ", stack.isEmpty())
    print("Adding 0 to 10 in the stack...")
    for i in range(10, 0, -1):
        stack.push(i)
    for i in range(1, 5):
        stack.push(i)
    print(stack)
    print("Stack size: ", stack.size())
    print("Stack peek and peekMinimum : ", stack.peek(),
        stack.peekMinimum())
    print("Pop...", stack.pop())
    print("Stack peek and peekMinimum : ", stack.peek(),
        stack.peekMinimum())
```

```
    print("Is the stack empty? ", stack.isEmpty())
    print(stack)
```

Set of Stacks

The following example implements a set of stacks, creating a new stack when the previous exceeds capacity. The `push` and `pop` methods are identical to a single stack:

```
[set_of_stacks.py]

from stack import Stack

class SetOfStacks(Stack):
    def __init__(self, capacity=4):
        self.setofstacks = []
        self.items = []
        self.capacity = capacity

    def push(self, value):
        if self.size() >= self.capacity:
            self.setofstacks.append(self.items)
            self.items = []
        self.items.append(value)

    def pop(self):
        value = self.items.pop()
        if self.isEmpty() and self.setofstacks:
            self.items = self.setofstacks.pop()
        return value

    def sizeStack(self):
        return len(self.setofstacks)*self.capacity + self.size()

    def __repr__(self):
        aux = []
        for s in self.setofstacks:
            aux.extend(s)
```

```
        aux.extend(self.items)
        return '{}'.format(aux)

if __name__ == '__main__':
    capacity = 5
    stack = SetOfStacks(capacity)
    print("Is the stack empty? ", stack.isEmpty())
    print("Adding 0 to 10 in the stack...")
    for i in range(10):
        stack.push(i)
    print(stack)
    print("Stack size: ", stack.sizeStack())
    print("Stack peek : ", stack.peek())
    print("Pop...", stack.pop())
    print("Stack peek: ", stack.peek())
    print("Is the stack empty? ", stack.isEmpty())
    print(stack)
```

Queues

Using a Queue to Model an Animal Shelter

```
[animal_shelter.py]

class Node(object):
    def __init__(self, animalName=None, animalKind=None,
        pointer=None):
        self.animalName = animalName
        self.animalKind = animalKind
        self.pointer = pointer
        self.timestamp = 0

class AnimalShelter(object):
    def __init__(self):
        self.headCat = None
        self.headDog = None
        self.tailCat = None
        self.tailDog = None
```

```
        self.animalNumber = 0

    def enqueue(self, animalName, animalKind):
        self.animalNumber += 1
        newAnimal = Node(animalName, animalKind)
        newAnimal.timestamp = self.animalNumber

        if animalKind == 'cat':
            if not self.headCat:
                self.headCat = newAnimal
            if self.tailCat:
                self.tailCat.pointer = newAnimal
            self.tailCat = newAnimal

        elif animalKind == 'dog':
            if not self.headDog:
                self.headDog = newAnimal
            if self.tailDog:
                self.tailDog.pointer = newAnimal
            self.tailDog = newAnimal

    def dequeueDog(self):
        if self.headDog:
            newAnimal = self.headDog
            self.headDog = newAnimal.pointer
            return str(newAnimal.animalName)
        else:
            return 'No Dogs!'

    def dequeueCat(self):
        if self.headCat:
            newAnimal = self.headCat
            self.headCat = newAnimal.pointer
            return str(newAnimal.animalName)
        else:
            return 'No Cats!'

    def dequeueAny(self):
        if self.headCat and not self.headDog:
            return self.dequeueCat()
```

```
        elif self.headDog and not self.headCat:
            return self.dequeueDog()
        elif self.headDog and self.headCat:
            if self.headDog.timestamp < self.headCat.timestamp:
                return self.dequeueDog()
            else:
                return self.dequeueCat()
        else:
            return ('No Animals!')

    def _print(self):
        print("Cats:")
        cats = self.headCat
        while cats:
            print(cats.animalName, cats.animalKind)
            cats = cats.pointer
        print("Dogs:")
        dogs = self.headDog
        while dogs:
            print(dogs.animalName, dogs.animalKind)
            dogs = dogs.pointer

if __name__ == '__main__':
    qs = AnimalShelter()
    qs.enqueue('bob', 'cat')
    qs.enqueue('mia', 'cat')
    qs.enqueue('yoda', 'dog')
    qs.enqueue('wolf', 'dog')
    qs._print()
    print("Deque one dog and one cat...")
    qs.dequeueDog()
    qs.dequeueCat()
    qs._print()
```

Priority Queues and Heaps

The example below uses Python's `heapq` package to find the N largest and smallest items in a sequence:

```
[find_N_largest_smallest_items_seq.py]

import heapq

def find_N_largest_items_seq(seq, N):
    return heapq.nlargest(N,seq)

def find_N_smallest_items_seq(seq, N):
    return heapq.nsmallest(N, seq)

def find_smallest_items_seq_heap(seq):
    heapq.heapify(seq)
    return heapq.heappop(seq)

def find_smallest_items_seq(seq):
   return min(seq)

def find_N_smallest_items_seq_sorted(seq, N):
    return sorted(seq)[:N]

def find_N_largest_items_seq_sorted(seq, N):
    return sorted(seq)[len(seq)-N:]

def test_find_N_largest_smallest_items_seq(module_name='this
    module'):
    seq = [1, 3, 2, 8, 6, 10, 9]
    N = 3
    assert(find_N_largest_items_seq(seq, N) == [10, 9, 8])
    assert(find_N_largest_items_seq_sorted(seq, N) == [8, 9, 10])
    assert(find_N_smallest_items_seq(seq, N) == [1,2,3])
    assert(find_N_smallest_items_seq_sorted(seq, N) == [1,2,3])
    assert(find_smallest_items_seq(seq) == 1)
    assert(find_smallest_items_seq_heap(seq) == 1)
```

The following example uses Python's `heapq` package to merge a two sorted sequences with little overhead:

```
[merge_sorted_seqs.py]
```

```
import heapq

def merge_sorted_seqs(seq1, seq2):
    result = []
    for c in heapq.merge(seq1, seq2):
        result.append(c)
    return result

def test_merge_sorted_seq(module_name='this module'):
    seq1 = [1, 2, 3, 8, 9, 10]
    seq2 = [2, 3, 4, 5, 6, 7, 9]
    seq3 = seq1 + seq2
    assert(merge_sorted_seq(seq1, seq2) == sorted(seq3))
```

Linked List

Find the kth Element from the End of a Linked List

```
[find_kth_from_the_end.py]

from linked_list_fifo import LinkedListFIFO
from node import Node

class LinkedListFIFO_find_kth(LinkedListFIFO):

    def find_kth_to_last(self, k):
        p1, p2 = self.head, self.head
        i = 0
        while p1:
            if i > k:
                try:
                    p2 = p2.pointer
                except:
                    break
            p1 = p1.pointer
            i += 1
        return p2.value
```

```
if __name__ == '__main__':
    ll = LinkedListFIFO_find_kth()
    for i in range(1, 11):
        ll.addNode(i)
    print('The Linked List:')
    print(ll._printList())
    k = 3
    k_from_last = ll.find_kth_to_last(k)
    print("The %dth element to the last of the LL of size %d is %d"
        %(k, ll.length, k_from_last))
```

Partitioning a Linked List in an Elements

This following function divides a linked list in a value, where everything smaller than this value goes to the front, and everything large goes to the back:

```
[part_linked_list.py]

from linked_list_fifo import LinkedListFIFO
from node import Node

def partList(ll, n):
    more = LinkedListFIFO()
    less = LinkedListFIFO()
    node = ll.head
    while node:
        item = node.value
        if item < n:
            less.addNode(item)
        elif item > n:
            more.addNode(item)
        node = node.pointer
    less.addNode(n)
    nodemore = more.head
    while nodemore:
        less.addNode(nodemore.value)
        nodemore = nodemore.pointer
```

```
    return less

if __name__ == '__main__':
    ll = LinkedListFIFO()
    l = [6, 7, 3, 4, 9, 5, 1, 2, 8]
    for i in l:
        ll.addNode(i)
    print('Before Part')
    ll._printList()
    print('After Part')
    newll = partList(ll, 6)
    newll._printList()
```

A Doubled Linked List FIFO

The function below implements a double-linked list:

```
[doubled_linked_list_fifo.py]

from linked_list_fifo import LinkedListFIFO

class dNode(object):
    def __init__(self, value=None, pointer=None, previous=None):
        self.value = value
        self.pointer = pointer
        self.previous = previous

class dLinkList(LinkedListFIFO):

    # print each node's value, starting from tail
    def printListInverse(self):
        node = self.tail
        while node:
            print(node.value)
            try:
                node = node.previous
            except:
                break
```

```
    # add a node in a position different from head,
    # ie, in the end of the list
    def _add(self, value):
        self.length += 1
        node = dNode(value)
        if self.tail:
            self.tail.pointer = node
            node.previous = self.tail
        self.tail = node

    # delete a node in some position
    def _delete(self, node):
        self.length -= 1
        node.previous.pointer = node.pointer
        if not node.pointer:
            self.tail = node.previous

    # locate node with some index
    def _find(self, index):
        node = self.head
        i = 0
        while node and i < index:
            node = node.pointer
            i += 1
        return node, i

    # delete nodes in general
    def deleteNode(self, index):
        if not self.head or not self.head.pointer:
            self._deleteFirst()
        else:
            node, i = self._find(index)
            if i == index:
                self._delete(node)
            else:
                print('Node with index {} not found'.format(index))

if __name__ == '__main__':
    from collections import Counter
```

```
    ll = dLinkList()
    for i in range(1, 5):
        ll.addNode(i)
    print('Printing the list...')
    ll._printList()
    print('Now, printing the list inversely...')
    ll.printListInverse()
    print('The list after adding node with value 15')
    ll._add(15)
    ll._printList()
    print("The list after deleting everything...")
    for i in range(ll.length-1, -1, -1):
        ll.deleteNode(i)
    ll._printList()
```

Summing Digits in in Linked Lists

Supposing two linked lists representing numbers, such that in each of their nodes they carry one digit. The function below sums the two numbers that these two linked lists represent, returning a third list representing the sum:

```
[sum_linked_list.py]

from linked_list_fifo import LinkedListFIFO
from node import Node

class LinkedListFIFOYield(LinkedListFIFO):
    # print each node's value, starting from the head
    def _printList(self):
        node = self.head
        while node:
            yield(node.value)
            node = node.pointer

def sumlls(l1, l2):
    lsum = LinkedListFIFOYield()
    dig1 = l1.head
    dig2 = l2.head
    pointer = 0
```

```
    while dig1 and dig2:
        d1 = dig1.value
        d2 = dig2.value
        sum_d = d1 + d2 + pointer
        if sum_d > 9:
            pointer = sum_d//10
            lsum.addNode(sum_d%10)
        else:
            lsum.addNode(sum_d)
            pointer = 0
        dig1 = dig1.pointer
        dig2 = dig2.pointer
    if dig1:
        sum_d = pointer + dig1.value
        if sum_d > 9:
            lsum.addNode(sum_d%10)
        else:
            lsum.addNode(sum_d)
        dig1 = dig1.pointer
    if dig2:
        sum_d = pointer + dig2.value
        if sum_d > 9:
            lsum.addNode(sum_d%10)
        else:
            lsum.addNode(sum_d)
        dig2 = dig2.pointer
    return lsum

if __name__ == '__main__':
    l1 = LinkedListFIFOYield() # 2671
    l1.addNode(1)
    l1.addNode(7)
    l1.addNode(6)
    l1.addNode(2)
    l2 = LinkedListFIFOYield() # 455
    l2.addNode(5)
    l2.addNode(5)
    l2.addNode(4)
    lsum = sumlls(l1, l2)
    l = list(lsum._printList())
```

```
    for i in reversed(l):
        print i
```

Find a Circular Linked List

The function below implements a method to see whether a linked list is circular, using two pointers with different paces:

```
[circular_linked_list.py]

from linked_list_fifo import LinkedListFIFO
from node import Node

class cicularLinkedListFIFO(LinkedListFIFO):
    def _add(self, value):
        self.length += 1
        node = Node(value, self.head)
        if self.tail:
            self.tail.pointer = node
        self.tail = node

def isCircularll(ll):
    p1 = ll.head
    p2 = ll.head
    while p2:
        try:
            p1 = p1.pointer
            p2 = p2.pointer.pointer
        except:
            break
        if p1 == p2:
            return True
    return False

if __name__ == '__main__':
    ll = LinkedListFIFO()
    for i in range(10):
        ll.addNode(i)
    ll._printList()
```

```
print(isCircularll(ll))
lcirc = cicularLinkedListFIFO()
for i in range(10):
    lcirc.addNode(i)
print(isCircularll(lcirc))
```

Chapter 9

Sorting

The simplest way to *sort* a group of items is to start by removing the smallest item from the group, and putting it in the first position. Then removing the next smallest item, and putting it as second, and so on.

This example is clearly an $\mathcal{O}(n^2)$ algorithm. It is obvious that we need better solutions! This is what this chapter is about. We are going to look at many sorting algorithms and we analyze their features and runtimes.

But first some nomenclature. An *in-place sort* does not use any additional memory to do the sorting (for example, swapping elements in an array). A *stable sort* preserves the relative order of otherwise identical data elements (for example, if two data elements have identical values, the one that was ahead of the other stays ahead). In any *comparison sort* problem, a *key* is the value (or values) that determines the sorting order. A comparison sort requires only that there is a way to determine if a key is less than, equal to, or greater than another key. Most sorting algorithms are comparison sorts where the worst-case running time for such sorts can be no better than $\mathcal{O}(n \ln n)$.

9.1 Quadratic Sort

Insertion Sort

Insertion sort is a simple sorting algorithm with average and worst runtime cases of $\mathcal{O}(n^2)$. It sorts by repeatedly inserting the next unsorted element in an initial sorted segment of the array.

For small data sets, it can be preferable to more advanced algorithms if the list is already sorted (it is a good way to add new elements to a presorted list):

```
[insertion_sort.py]

def insertion_sort(seq):
    for i in range(1, len(seq)):
        j = i
        while j > 0 and seq[j-1] > seq[j]:
            seq[j-1], seq[j] = seq[j], seq[j-1]
            j -= 1
    return seq

def insertion_sort_rec(seq, i = None):
    if i == None:
      i = len(seq) -1
    if i == 0:
      return i
    insertion_sort_rec(seq, i-1)
    j = i
    while j > 0 and seq[j-i] > seq[j]:
        seq[j-1], seq[j] = seq[j], seq[j-1]
        j -= 1
    return seq
```

Selection Sort

Selection sort is based on finding the smallest or largest element in a list and exchanging it to the first, then finding the second, etc, until the end is reached. Even when the list is sorted, it is $\mathcal{O}(n^2)$ (and not stable):

```
[selection_sort.py]

def selection_sort(seq):
    for i in range(len(seq) -1, 0, -1):
        max_j = i
        for j in range(max_j):
```

```
            if seq[j] > seq[max_j]:
                max_j = j
            seq[i], seq[max_j] = seq[max_j], seq[i]
    return seq
```

A more sophisticated version of this algorithm is called **quick select** and it is presented in the next chapter. Quick sort is used to find the kth element of an array, and its median.

Gnome Sort

Gnome sort works by moving forward to find a misplaced value and then moving backward to place it in the right position:

```
[gnome_sort.py]

def gnome_sort(seq):
    i = 0
    while i < len(seq):
        if i ==0 or seq[i-1] <= seq[i]:
            i += 1
        else:
            seq[i], seq[i-1] = seq[i-1], seq[i]
            i -= 1
    return seq
```

9.2 Linear Sort

Count Sort

Count sort sorts integers with a small value range, counting occurrences and using the cumulative counts to directly place the numbers in the result, updating the counts as it goes.

There is a loglinear limit on how fast you can sort if all you know about your data is that they are greater or less than each other. However, if you can also count events, sort becomes linear in time, $\mathcal{O}(n + k)$:

```
[count_sort.py]

from collections import defaultdict

def count_sort_dict(a):
    b, c = [], defaultdict(list)
    for x in a:
        c[x].append(x)
    for k in range(min(c), max(c) + 1):
        b.extend(c[k])
    return b
```

If several values have the same key, they will have the original order with respect with each other, so the algorithm is *stable*.

9.3 Loglinear Sort

The `sort()` and `sorted()` Methods

In Python, we would normally sort a list with `list.sort()`, which is an in-place method. Additionally, any other iterable item can be sorted with the function. They are both a very efficient implementation of the Python's `timsort` algorithm[1].

The `sorted()` function can also be customized though optional arguments, for example: `reverse=True`; `key=len`; `str.lower` (treating uppercase and lowercase the same); or even with a custom sorting function.

Merge Sort

Merge sort divides the list in half to create two unsorted lists. These two unsorted lists are sorted and merged by continually calling the merge-sort algorithm, until you get a list of size 1.

[1] Timsort is a hybrid sorting algorithm, derived from merge sort and insertion sort, and invented by Tim Peters for Python.

Merge sort is stable, as well as fast, for large data sets. However, since it is not in-place, it requires much more memory than many other algorithms. The space complexity is $\mathcal{O}(n)$ for arrays and $\mathcal{O}(\ln n)$ for linked lists[2]. The best, average, and worst case times are all $\mathcal{O}(n \ln n)$.

Merge sort is a good choice when the data set is too large to fit into the memory. The subsets can be written to disk in separate files until they are small enough to be sorted in memory. The merging is easy, as it involves reading single elements at a time from each file and writing them to the final file in the correct order:

```
[merge_sort.py]

def merge_sort(seq):
    '''
    >>> seq = [3, 5, 2, 6, 8, 1, 0, 3, 5, 6, 2]
    >>> merge_sort(seq)
    [0, 1, 2, 2, 3, 3, 5, 5, 6, 6, 8]
    '''
    if len(seq) < 2:
        return seq
    mid = len(seq)//2
    lft, rgt = seq[:mid], seq[mid:]
    if len(lft)>1:
        lft = merge_sort(lft)
    if len(rgt)>1:
        rgt = merge_sort(rgt)
    res = []
    while lft and rgt:
        if lft [-1]>= rgt[-1]:
            res.append(lft.pop())
        else:
            res.append(rgt.pop())
    res.reverse()
    return(lft or rgt) + res
```

We can also write this algorithm by separating the merge from the sort part:

```
def merge_sort_sep(seq):
```

[2]Never ever consider to sort a linked list!

```
    '''
    >>> seq = [3, 5, 2, 6, 8, 1, 0, 3, 5, 6, 2]
    >>> merge_sort_sep(seq)
    [0, 1, 2, 2, 3, 3, 5, 5, 6, 6, 8]
    '''
    if len(seq) < 2 :
        return seq
    mid = len(seq)//2
    left = merge_sort(seq[:mid])
    # notice that mid is included!
    right = merge_sort(seq[mid:])
    return merge(left, right)

def merge(left, right):
    if not left or not right:
        return left or right # nothing to be merged
    result = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    if left[i:] : result.extend(left[i:])
    if right[j:] : result.extend(right[j:])
    return result
```

Quick Sort

Quick sort works by choosing a *pivot* and partitioning the array so that the elements that are smaller than the pivot go to the left, and elements that are larger go to the right. Then, it recursively sorts these left and right parts.

The choice of the pivot value is a key to the performance. It can be shown that always choosing the value in the middle of the set is the best choice for already-sorted data and no worse than most other choices for random

unsorted data.

The wort case runtime is $\mathcal{O}(n^2)$ in the rare cases when partitioning keeps producing a region of $n-1$ elements (when the pivot is the minimum or the maximum value). The best case produces two $n/2$-sized lists, being $\mathcal{O}(n \ln n)$. This algorithm is not stable.

Here is an example of quick sort in Python:

```
[quick_sort.py]

def quick_sort(seq):
    if len(seq) < 2: return seq
    ipivot = len(seq)//2
    pivot = seq[ipivot]
    before = [x for i,x in enumerate(seq) if x <= pivot and i !=
        ipivot]
    after = [x for i,x in enumerate(seq) if x > pivot and i !=
        ipivot]
    return qs(before) + [pivot] + qs(after)
```

And here if we divide the algorithm into partition and sort:

```
def partition(seq):
    pi,seq = seq[0],seq[1:]
    lo = [x for x in seq if x <= pi]
    hi = [x for x in seq if x > pi]
    return lo, pi, hi

def quick_sort_divided(seq):
    if len(seq) < 2: return seq
    lo, pi, hi = partition(seq)
    return quick_sort_divided(lo) + [pi] + quick_sort_divided(hi)
```

Heap Sort

Heap sort is similar to a selection sort, except that the unsorted region is a heap, so finding the largest element n times results in a loglinear runtime.

In a heap, for every node other than the root, it has at least (at most) the value of its parent. Thus, the smallest (largest) element is stored at the

root and the subtrees rooted at a node contain only larger (smaller) values than does the node itself.

Although the insertion is only $\mathcal{O}(1)$, the performance of validating (the heap order) is $\mathcal{O}(\ln n)$. Searching (traversing) is $\mathcal{O}(n)$.

In Python, a heap sort can be implemented by pushing all values onto a heap and then popping off the smallest values one at a time:

```
[heap_sort1.py]

import heapq

def heap_sort1(seq):
    h = []
    for value in seq:
        heapq.heappush(h, value)
    return [heapq.heappop(h) for i in range(len(h))]
```

9.4 Comparison Between Sorting Methods

Algorithm	Data Structure	Time Complexity			Worst Case Auxiliary Space Complexity
		Best	Average	Worst	Worst
Quicksort	Array	O(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Mergesort	Array	O(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Array	O(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Array	Ω(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	Array	Ω(n)	O(n^2)	O(n^2)	O(1)
Select Sort	Array	O(n^2)	O(n^2)	O(n^2)	O(1)
Bucket Sort	Array	Ω(n+k)	Θ(n+k)	O(n^2)	O(nk)
Radix Sort	Array	Ω(nk)	Θ(nk)	O(nk)	O(n+k)

9.5 Additional Exercises

Quadratic Sort

The following program implements a *bubble sort*, a very inefficient sorting algorithm:

```
[bubble_sort.py]

def bubble_sort(seq):
    size = len(seq) -1
    for num in range(size, 0, -1):
        for i in range(num):
            if seq[i] > seq[i+1]:
                temp = seq[i]
                seq[i] = seq[i+1]
                seq[i+1] = temp
    return seq
```

Linear Sort

The example below shows a simple count sort for people ages:

```
[counting_sort.py]

def counting_sort_age(A):
    oldestAge = 100
    timesOfAge = [0]*oldestAge
    ageCountSet = set()
    B = []
    for i in A:
        timesOfAge[i] += 1
        ageCountSet.add(i)
    for j in ageCountSet:
        count = timesOfAge[j]
        while count > 0:
            B.append(j)
            count -= 1
    return B
```

The example below uses *quick sort* to find the k largest elements in a sequence:

```
[find_k_largest_seq_quicksort.py]

import random

def swap(A, x, y):
    tmp = A[x]
    A[x] = A[y]
    A[y] = tmp

def qselect(A, k, left=None, right=None):
    left = left or 0
    right = right or len(A) - 1
    pivot = random.randint(left, right)
    pivotVal = A[pivot]
    swap(A, pivot, right)
    swapIndex, i = left, left
```

```
    while i <= right - 1:
        if A[i] < pivotVal:
            swap(A, i, swapIndex)
            swapIndex += 1
        i += 1
    swap(A, right, swapIndex)
    rank = len(A) - swapIndex
    if k == rank:
        return A[swapIndex]
    elif k < rank:
        return qselect(A, k, left=swapIndex+1, right=right)
    else:
        return qselect(A, k, left=left, right=swapIndex-1)

def find_k_largest_seq_quickselect(seq, k):
    kth_largest = qselect(seq, k)
    result = []
    for item in seq:
        if item >= kth_largest:
            result.append(item)
    return result
```

Chapter 10

Searching

The most common *searching* algorithms are the *sequential search* and the *binary search*. If an input array is not *sorted*, or the input elements are accommodated by dynamic containers (such as linked lists), the search is usually sequential. If the input is a sorted array, the binary search algorithm is the best choice. If we are allowed to use auxiliary memory, a hash table might help the search, with which a value can be located in $\mathcal{O}(1)$ time (with a key).

10.1 Unsorted Arrays

10.1.1 Sequential Search

In the following example we illustrate the runtime of a sequential search for items stored in a collection. If the item is present, the best case is $\mathcal{O}(1)$, the average case is $\mathcal{O}(n/2)$, and the worst case is $\mathcal{O}(n)$. However, if the item is not present, all three cases will be $\mathcal{O}(n)$:

```
[sequential_search.py]

def sequential_search(seq, n):
    for item in seq:
        if item == n:
         return True
    return False
```

Now, if we sort the sequence first, we can improve the sequential search in the case when the item is not present to have the same runtimes as when the item is present:

```
[ordered_sequential_search.py]

def ordered_sequential_search(seq, n):
    item = 0
    for item in seq:
        if item > n:
         break
        if item == n:
         return True
    return False
```

10.1.2 Quick Select and Order Statistics

We can use an adapted version of **quick sort** to find the **kth smallest number** in a list.

Such a number is called the **kth order statistic** and this search includes the cases of finding the minimum, maximum, and median elements.

Quick select is $\mathcal{O}(n)$ in the worst case because we only look to one side of the array in each iteration: $\mathcal{O}(n) = \mathcal{O}(n) + \mathcal{O}(n/2) + \mathcal{O}(n/4)...$

Sublinear performance is possible for structured data. For instance, we can achieve $\mathcal{O}(1)$ for an array of sorted data.

```
[quick_select.py]

def quickSelect(seq, k):
    len_seq = len(seq)
    if len_seq < 2:
      return seq
    ipivot = len_seq // 2
    pivot = seq[ipivot]
    smallerList = [x for i,x in enumerate(seq) if x <= pivot and i
        != ipivot]
    largerList = [x for i,x in enumerate(seq) if x > pivot and i !=
        ipivot]
```

```
    # here the different part starts
    m = len(smallerList)
    if k == m:
        return pivot
    elif k < m:
        return quickSelect(smallerList, k)
    else:
        return quickSelect(largerList, k-m-1)
```

In general, we can define the median as the *the value that is bigger than half of the array*. This algorithm is important in the context of larger problems such as finding the **nearest neighbor** or the **shortest path**.

10.2 Sorted Arrays

10.2.1 Binary Search

A binary search finds the position of a specified input value (the key) within a sorted array. The runtime is $\mathcal{O}(\ln n)$:

In each step, the algorithm compares the search item with the middle element of the array. If they match the item is returned. Otherwise, if the item is smaller or larger, the process is repeated until exhaustion.

A recursive examples is the following:

```
[binary_search.py]

def binary_search_rec(seq,key):
    if not seq :
       return False
    mid = len(seq)//2
    if key == seq[mid] :
      return True
    elif key < seq[mid] :
      return binary_search_rec(seq[:mid],key)
    else :
      return binary_search_rec(seq[mid+1:],key)
```

An iterative example is the following:

```
def binary_search_iter(seq,key):
    hi, lo = len(seq), 0
    while lo < hi :
        mid = (hi + lo)//2
        if key == seq[mid] : return True
        elif key < seq[mid] : hi = mid
        else : lo = mid + 1
    return False
```

The `bisect` Module

Python's built-in `bisect()` module is available for binary search in a sorted sequence:

```
>>> from bisect import bisect
>>> list = [0, 3, 4, 5]
>>> bisect(list, 5)
 4
```

Note that the module returns the index *after* the key, which is *where you should place the new value*. Other available functions are `bisect_right` and `bisect_left`.

10.3 Additional Exercises

Searching in a Matrix

The following module searches an entry in a matrix where the rows and columns are sorted. In this case, every row is increasingly sorted from left to right, and every column is increasingly sorted from top to bottom. The runtime is linear on $\mathcal{O}(m+n)$:

```
[search_entry_matrix.py]

def find_elem_matrix_bool(m1, value):
    found = False
    row = 0
    col = len(m1[0]) - 1
    while row < len(m1) and col >= 0:
        if m1[row][col] == value:
            found = True
            break
        elif m1[row][col] > value:
            col-=1
        else:
            row+=1
    return found
```

The following problem searches an element in a matrix where in every row, the values increase from left to right (but the last number in a row is smaller than the first number in the next row).

The naive brute force solution scans all numbers and costs $\mathcal{O}(mn)$. However, since the numbers are already sorted, the matrix can be viewed as an 1D sorted array and we can use a binary search algorithm with efficiency $\mathcal{O}(lognm)$:

```
[searching_in_a_matrix.py]

import numpy

def searching_in_a_matrix(m1, value):
    rows = len(m1)
    cols = len(m1[0])
```

```
    lo = 0
    hi = rows*cols
    while lo < hi:
        mid = (lo + hi)//2
        row = mid//cols
        col = mid%cols
        v = m1[row][col]
        if v == value:
         return True
        elif v > value:
         hi = mid
        else:
         lo = mid+1
    return False
```

Unimodal Arrays

An array is *unimodal* if it is consisted of an increasing sequence followed by a decreasing sequence.

The example below shows how to find the "locally maximum" of an array using binary search:

```
[find_max_unimodal_array.py]

def find_max_unimodal_array(A):
    if len(A) <= 2 :
      return None
    left = 0
    right = len(A)-1
    while right > left +1:
        mid = (left + right)//2
        if A[mid] > A[mid-1] and A[mid] > A[mid+1]:
            return A[mid]
        elif A[mid] > A[mid-1] and A[mid] < A[mid+1]:
            left = mid
        else:
            right = mid
    return None
```

Calculating Square Roots

The example below implements the square root of a number using binary search:

```
[find_sqrt_bin_search.py]

def find_sqrt_bin_search(n, error=0.001):
    lower = n < 1 and n or 1
    upper = n < 1 and 1 or n
    mid = lower + (upper - lower) / 2.0
    square = mid * mid
    while abs(square - n) > error:
        if square < n:
            lower = mid
        else:
            upper = mid
        mid = lower + (upper - lower) / 2.0
        square = mid * mid
    return mid
```

Counting Frequency of Elements

The following example finds how many times a k element appears in a sorted list. Since the array is sorted, binary search gives a $\mathcal{O}(logn)$ runtime:

```
[find_time_occurence_list.py]

from binary_search import binary_search

def find_time_occurrence_list(seq, k):
    index_some_k = binary_serch(seq, k)
    count = 1
    sizet = len(seq)
    for i in range(index_some_k+1, sizet): # go up
        if seq[i] == k:
```

```
            count +=1
        else:
            break
    for i in range(index_some_k-1, -1, -1): # go down
        if seq[i] == k:
            count +=1
        else:
            break
    return count
```

Intersection of Arrays

The snippet below shows three ways to perform the intersection of two sorted arrays.

The simplest way is to use sets, however this will not preserve the ordering.

The second example uses an adaptation of the merge sort.

The third example is suitable when one of the arrays is much larger than other. In this case, binary search is the best option:

```
[intersection_two_arrays.py]

def intersection_two_arrays_sets(seq1, seq2):
    set1 = set(seq1)
    set2 = set(seq2)
    return set1.intersection(set2) #same as list(set1 & set2


def intersection_two_arrays_ms(seq1, seq2):
    res = []
    while seq1 and seq2:
        if seq1[-1] == seq2[-1]:
            res.append(seq1.pop())
            seq2.pop()
        elif seq1[-1] > seq2[-1]:
            seq1.pop()
        else:
            seq2.pop()
```

```
    res.reverse()
    return res


from binary_search import binary_search
def intersection_two_arrays_bs(seq1, seq2):
    if len(seq1) > len(seq2):
      seq, key = seq1, seq2
    else:
      seq, key = seq2, seq1
    intersec = []
    for item in key:
        if binary_search(seq, item):
            intersec.append(item)
    return intersec
```

Chapter 11

Dynamic Programming

Dynamic programming is used to simplify a complicated problem by breaking it down into simpler subproblems with recursion. If a problem has an *optimal substructure* and *overlapping subproblems*, it may be solved by dynamic programming.

Optimal substructure means that the solution to a given optimization problem can be obtained by a combination of optimal solutions to its subproblems.

When a problem has *overlapping subproblems*, these are solved once and then their solutions are stored for future retrieval.

11.1 Memoization

Dynamically Solving the Fibonacci Series

High-level languages such as Python can implement the recursive formulation directly, *caching* return values. *Memoization* is a method which for each call made more than once (with the same arguments), the result is returned directly from the *cache*.

For example, we can dynamically solve the exponential Fibonacci series. For this, we use a `memo` function that uses nested scopes to give the wrapped function memory:

```
[memo.py]
```

```
from functools import wraps

def memo(func):
    cache = {}
    @wraps(func)
    def wrap(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrap

>>> fibonacci = memo(fibonacci)
>>> fibonacci(130)
1066340417491710595814572169
```

We could even use the decorator directly in the function:

```
@memo
def fib(i):
    if i < 2: return 1
    return fib(i-1) + fib(i-2)
>>> fibonacci(130)
1066340417491710595814572169
```

Longest Increasing Subsequence

Another interesting application of memoization is the problem of finding the **longest increasing subsequence** of a sequence.

A naive recursive solution gives a $\mathcal{O}(2^n)$ runtime. However, the iterative solution can be solved in loglinear time using dynamic programming.

For instance, in the example below we benchmark both solutions with an array with 40 elements. We find that the memoized version takes less than one second to run, while the solution without dynamic programming, takes over 120 seconds:

```
[memoized_longest_inc_subseq.py]

from itertools import combinations
from memo import memo
```

```
def naive_longest_inc_subseq(seq):
    for length in range(len(seq), 0, -1):
        for sub in combinations(seq, length):
            if list(sub) == sorted(sub):
                return len(sub)

def memoized_longest_inc_subseq(seq):
    @memo
    def L(cur):
        res = 1
        for pre in range(cur):
            if seq[pre] <= seq[cur]:
                res = max(res, 1 + L(pre))
        return res
    return max(L(i) for i in range(len(seq)))
```

Part III

Climbing some beautiful Graphs and Trees!

Chapter 12

Introduction to Graphs

12.1 Basic Definitions

A *graph* is an abstract network, consisting of *nodes* (or vertices, V) connected by *edges* (or arcs, E).

A graph can be defined as a pair of sets, $G = (V, E)$, where the node set V is any finite set, and the edge set E is a set of node pairs.

For example, some graph can be simply represented by the node set $V = \{a, b, c, d\}$ and the edge set $E = \{\{a, b\}, \{b, c\}, \{c, d\}, \{d, a\}\}$.

Direction of a Graph

If a graph has no direction, it is referred as *undirected*. In this case, nodes with an edge between them are *adjacent* and adjacent nodes are *neighbors*.

If the edges have a direction, the graph is *directed* (digraph). In this case, the graph has *leaves* and the edges are no longer unordered. For instance, an edge between nodes u and v is either an edge (u, v) from u to v, or an edge (v, u) from v to u.

Additionally, in a digraph G, the function $E(G)$ is a relation over $V(G)$.

Subgraphs

A *subgraph* of G consists of a subset of V and E. A *spanning subgraph* contains all the nodes of the original graph.

Completeness of a Graph

If all the nodes in a graph are pairwise adjacent, the graph is called *complete*.

Degree in a Node

The number of undirected edges incident on a node is called *degree*. Zero-degree graphs are called *isolated*.

For directed graphs, we can split this number into *in-degree* (incoming edges/parents) and *out-degree/children* (outgoing edges).

Paths, Walks, and Cycle

A *path* in G is a subgraph where the edges connect the nodes in a *sequence*, without revisiting any node. In a directed graph, a path has to follow the directions of the edges.

A *walk* is an alternating sequence of nodes and edges that allows nodes and edges to be visited multiple times.

A *cycle* is like a path except that the last edge links the last node to the first.

Length of a Path

The *length* of a path or walk is the value given by its edge count.

Weight of an Edge

Associating *weights* with each edge in G gives us a *weighted graph*. The weight of a path or cycle is the sum of its edge weights. So, for unweighted graphs, it is simply the number of edges.

Planar Graphs

A graph that can be drawn on the plane without crossing edges is called *planar*. This graph has *regions*, which are areas bounded by the edges.

Interestingly, the *Euler's formula* for connected planar graphs says that $V - E + F = 2$, where V, E, F are the number of nodes, edges, and regions, respectively.

Graph Traversal

A *traversal* is a walk through all the connected components of a graph. The main difference between graph traversals is the ordering of the *to-do* list among the unvisited nodes that have been discovered.

Connected and Strongly Connected Components

An undirected graph is *connected* if there is a path from every node to every other node. A directed graph is *connected* if its underlying undirected graph is connected.

A *connected component* is a maximal subgraph that is connected. Connected components can be found using *traversal algorithms* such as *depth-first searching* (DFS) or *breath-first searching* (BFS), as we will see in following chapters.

If there is a path from every node to every other node in a directed graph, the graph is called *strongly connected*. A *strongly connected component* (SCC) is a maximal subgraph that is strongly connected.

Trees and Forests

A *forest* is a *cycle-free* graph. A *tree* is an acyclic, connected, and directed graph. A forest consists of one of more trees.

In a tree, any two nodes are connected by exactly one path. Adding any new edge to it creates a cycle and removing any edge yields unconnected components.

12.2 The Neighborhood Function

A graph's *neighborhood function*, $N(V)$, is a container (or iterable object) of all neighbors of V.

The most well-known data structures used to represent them are *adjacent lists* and *adjacent matrices*.

Adjacent Lists

For each node in an *adjacent list*, we have access to a list (or set or container or iterable) of its neighbor.

Supposing we have n nodes, each adjacent (or neighbor) list is just a list of such numbers. We place the lists into a main list of size n, indexable by the node numbers, where the order is usually arbitrary.

Using Sets as Adjacent Lists

We can use Python's set type to implement adjacent lists:

```
>>> a,b,c,d,e,f = range(6) # nodes
>>> N = [{b,c,d,f}, {a,d,f}, {a,b,d,e}, {a,e}, {a,b,c}, {b,c,d,e}]
>>> b in N[a] # membership
True
>>> b in N[b] # membership
False
>>> len(N[f]) # degree
4
```

Using Lists as Adjacent Lists

We can also use Python's lists to implement adjacent lists, which let you efficiently iterate $N(V)$ over any node V. Replacing sets with lists makes membership checking to be $\mathcal{O}(n)$.

If all that your algorithm does is *iterating over neighbors*, using list may be preferential. However if the graph is dense (many edges), adjacent sets are a better solution:

```
>>> a,b,c,d,e,f = range(6) # nodes
>>> N = [[b,c,d,f], [a,d,f], [a,b,d,e], [a,e], [a,b,c], [b,c,d,e]]
>>> b in N[a] # membership
True
>>> b in N[b] # membership
False
>>> len(N[f]) # degree
4
```

Deleting objects from the middle of a Python list is $\mathcal{O}(n)$, but deleting from the end is only $\mathcal{O}(1)$. If the order of neighbors is not important, you can delete an arbitrary neighbor in $\mathcal{O}(1)$ time by swapping it in to the last item in the list and then calling `pop()`.

Using Dictionaries as Adjacent Lists

We can also use dictionaries as adjacent lists. In this case, the neighbors would be the keys, and we are able to associate each of them with some extra value, such as an edge weight:

```
>>> a,b,c,d,e,f = range(6) # nodes
>>> N = [{b:2,c:1,d:4,f:1}, {a:4,d:1,f:4}, {a:1,b:1,d:2,e:4},
   {a:3,e:2}, {a:3,b:4,c:1}, {b:1,c:2,d:4,e:3}]
>>> b in N[a] # membership
True
>>> len(N[f]) # degree
4
>>> N[a][b] # edge weight for (a,b)
2
```

A more flexible approach for node labels is to use dictionaries as a main structure only. For instance, we can use a dictionary with adjacency sets:

```
>>> a,b,c,d,e,f = range(6) # nodes
>>> N = { 'a':set('bcdf'), 'b':set('adf'), 'c':set('abde'),
   'd':set('ae'), 'e':set('abc'), 'f':set('bcde')}
>>> 'b' in N['a'] # membership
True
>>> len(N['f']) # degree
4
```

Adjacent Matrices

In *adjacent matrices*, instead of listing all the neighbors for each node, we have one row with one position for each possible neighbor, filled with `True` and `False` values.

The simplest implementation of adjacent matrices is given by nested lists. Note that the diagonal is always `False`:

```
>>> a,b,c,d,e,f = range(6) # nodes
>>> N = [[0,1,1,1,0,1], [1,0,0,1,0,1], [1,1,0,1,1,0],
   [1,0,0,0,1,0], [1,1,1,0,0,0], [0,1,1,1,1,0]]
>>> N[a][b] # membership
1
>>> N[a][e]
0
>>> sum(N[f]) # degree
4
```

An adjacent matrix for an *undirected graph will always be symmetric*. To add weight to adjacent matrices, we can replace `True` and `False` by values. In this case, non-existent edges can be represented by infinite weights (`float('inf')`, or `None`, -1, or very large values):

```
>>> _ = float('inf') # nodes
>>> N = [[_,2,1,4,_,1], [4,_,_,1,_,4], [1,1,_,2,4,_],
   [3,_,_,_,2,_], [3,4,1,_,_,_], [1,2,_,4,3,_]]
>>> N[a][b] < _ # membership
True
>>> sum(1 for w in N[f] if w < _) # degree
4
```

Looking up an edge in an adjacent matrix is $\mathcal{O}(1)$ while iterating over a node's neighbor is $\mathcal{O}(n)$.

12.3 Connection to Trees

While in a graph there may be multiple references to any given node, in a *tree* each node (data element) is referenced only by at most one other node, the *parent node*.

The *root* node is the node that has no parent. The nodes referenced by a parent node are called *children*.

A tree is said to be *full* and *complete* if all of its leaves are at the bottom and all of the non-leaf nodes have exactly two children.

Height or Depth of a Tree

The *height* of a tree is the length of the path from the root to the deepest node in the tree (it is the maximum level of any node in this tree).

If the height of a tree is represented as the log of the number of leaves, the integer number from the log will, redundantly, be also called depth.

Level or Depth of a Node

The *level* of a node is the length of path from the root to this node.

Representing Trees

The simplest way of representing a tree is by a nested lists:

```
>>> T = ['a', ['b', ['d', 'f']], ['c', ['e', 'g']]]
>>> T[0]
'a'
>>> T[1][0]
'b'
>>> T[1][1][0]
'd'
>>> T[1][1][1]
'f'
>>> T[2][0]
'c'
>>> T[2][1][1]
'g'
```

However, this becomes very hard to handle if we simply add a couple more branches. The best solution is representing trees as a collection of nodes.

Let us start with a simple example, where we define a simple tree class with an attribute for value, another for a children (or 'next'), and a method for printing the result:

```
[tree.py]

class SimpleTree(object):
    def __init__(self, value=None, children = None):
```

```
        self.value = value
        self.children = children
        if self.children == None:
            self.children = []

    def __repr__(self, level=0):
        ret = "\t"*level+repr(self.value)+"\n"
        for child in self.children:
            ret += child.__repr__(level+1)
        return ret

def main():
    """
    'a'
       'b'
          'd'
          'e'
       'c'
          'h'
          'g'
    """
    st = SimpleTree('a', [SimpleTree('b', [SimpleTree('d'),
        SimpleTree('e')] ), SimpleTree('c', [SimpleTree('h'),
        SimpleTree('g')]) ])
    print(st)
```

In the next chapter we will learn how to improve this class, including many features and methods that a tree can hold.

Chapter 13

Binary Trees

Binary trees are tree data structures where each node has at most two child nodes: the *left* and the *right*. Child nodes may contain references to their parents. The *root* of a tree (the ancestor of all nodes) can exist either inside or outside the tree.

The *degree* of every node is at most two. Supposing that an arbitrary rooted tree has m internal nodes and each internal node has exactly two children, if the tree has n leaves, the degree of the tree is $n - 1$:

$$2m = n + m - 1 \rightarrow m = n - 1,$$

i.e a tree with n nodes has exactly $n - 1$ branches or degree.

Representing Binary Trees

A natural way to represent binary trees is with a collection of nodes.

A node in a binary tree should carry attributes for the item's value and pointers for left and right children. It also should have methods such as search and add:

```
[binary_tree.py]

class Node(object):
    def __init__(self, item=None,):
        self.item = item
        self.left = None
```

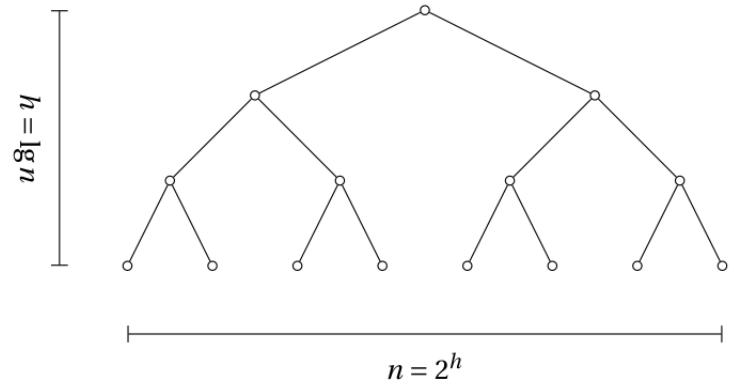


Figure 13.1: The height (h) and width (number of leaves) of a (*perfectly balanced*) binary tree.

```
        self.right = None

    def __repr__(self):
        return '{}'.format(self.item)

    def _add(self, value):
        new_node = Node(value)
        if not self.item:
            self.item = new_node
        elif not self.left:
            self.left = new_node
        elif not self.right:
            self.right = new_node
        else:
            self.left = self.left._add(value)
        return self

    def _search(self, value):
        if self.item == value:
            return True # or self
        found = False
        if self.left:
            found = self.left._search(value)
```

```
        if self.right:
            found = found or self.right._search(value)
        return found

    def _isLeaf(self):
        return not self.right and not self.left
```

A binary tree can be represented as a class that holds nodes. It should have an attribute for root and methods such as tree-printing:

```
class BT(object):
    def __init__(self):
        self.root = None

    def add(self, value):
        if not self.root:
            self.root = Node(value)
        else:
            self.root._add(value)

    def printPreorder(self):
        current = self.root
        nodes, stack = [], []
        while stack or current:
            if current:
                nodes.append(current.item)
                stack.append(current)
                current = current.left
            else:
                current = stack.pop()
                current = current.right
        print nodes

    def search(self, value):
        if self.root:
            return self.root._search(value)
```

13.1 Binary Search Trees

A *binary search tree* (BST) is a binary tree that keeps items in a sorted order. For a BST to be valid, each node's value should be greater than any element in the leftmost subtree and smaller than any element in rightmost subtree. Additionally, there must be no duplicate nodes.

If the binary search tree is balanced, the following operations are $\mathcal{O}(\ln n)$: (i) finding a node with a given value (lookup), (ii) finding a node with maximum or minimum value, and (iii) insertion or deletion of a node.

Representing Binary Search Trees

The following snippet implements a node class for a binary search tree. The main difference from a simple binary tree is when we insert a new node or search for some node's value:

```
[binary_search_tree.py]

class Node(object):
   (...)
    def _add(self, value):
        new_node = Node(value)
        if not self.item:
            self.item = new_node
        else:
            if value > self.item:
                self.right = self.right and self.right._add(value)
                    or new_node
            elif value < self.item:
                self.left = self.left and self.left._add(value) or
                    new_node
            else:
                print("BSTs do not support repeated items.")
        return self

    def _search(self, value):
        if self.item == value:
            return True # or self
        elif self.left and value < self.item:
                return self.left._search(value)
```

```
        elif self.right and value > self.item:
                return self.right._search(value)
        else:
            return False
```

13.2 Self-Balancing BSTs

A *balanced* tree is a tree where the differences of the height of every subtree is at most equal to 1. A *self-balancing* binary search tree is any node-based binary search tree that automatically keeps itself balanced.

By applying a balance condition we ensure that the worst case runtime of common tree operations will be at most $\mathcal{O}(\ln n)$.

A *balancing factor* can be attributed to each internal node in a tree, being the difference between the heights of the left and right subtrees. There are many balancing methods for trees, but they are usually based on two operations:

- ⋆ *Node splitting (and merging)*: nodes are not allowed to have more than two children, so when a node become overfull it splits into two subnodes.
- ⋆ *Node rotations*: process of switching edges. If x is the parent of y, we make y the parent of x and x must take over one of the children of y.

AVL Trees

An *AVL tree* is a binary search tree with a *self-balancing condition* where the difference between the height of the left and right subtrees cannot be more than one.

To implement an AVL tree, we can start by adding a self-balancing method to our BST classes, called every time we add a new node to the tree. The method works by continuously checking the height of the tree, which is added as a new attribute.

Red-black Trees

Red-black trees are an evolution of a binary search trees that aim to keep the tree balanced without affecting the complexity of the primitive operations.

This is done by coloring each node in the tree with either red or black and preserving a set of properties that guarantees that the deepest path in the tree is not longer than twice the shortest one.

Red-black trees have the following properties:

- ⋆ Every node is colored with either red or black.
- ⋆ All leaf (nil) nodes are colored with black; if a node's child is missing then we will assume that it has a nil child in that place and this nil child is always colored black.
- ⋆ Both children of a red node must be black nodes.
- ⋆ Every path from a node n to a descendant leaf has the same number of black nodes (not counting node n). We call this number the *black height* of n.

Binary Heaps

Binary heaps are *complete* balanced binary trees. The heap property makes it easier to maintain the structure, *i.e.,* the balance of the tree.

In a heap, there is no need to modify the tree structure by splitting or rotating nodes: the only operations needed is swapping (both parents and child nodes).

In a binary heap, considering a node at index i, we have that:

- ⋆ the parent index is $\frac{i-1}{2}$,
- ⋆ the left child index is $2i + 1$,
- ⋆ the right child index is $2i + 2$.

13.3 Additional Exercises

Check BST's Largest Item

```
[check_largest_item.py]

def largest(node):

    if node.right:
        return largest(node.right)
    return node.item
```

Check if the Tree is Balanced

```
[check_if_balanced.py]

def isBalanced(node, left=0, right=0):
    if not node:
        return (left - right) < 2

    return isBalanced(node.left, left+1, right) and \
        isBalanced(node.right, left, right+1)
```

Check if Tree is BST

```
[check_if_bst.py]

def isBST(node, min_node=float("-infinity"),
    maxVal=float("infinity")):
    if not node:
        return True

    if not min_node <= node.item <= maxVal:
        return False

    return isBST(node.left, min_node, node.item) and \
          isBST(node.right, node.item, maxVal)
```

Chapter 14

Traversing Trees

Traversals are algorithms used to visit the objects (nodes) in some connected structure, such as a tree or a graph. Traversal problems can be either visiting *every node* or visiting only *some specific nodes*.

14.1 Depth-First Search

Depth-first traversals, or *depth-first search* (DFS), are algorithms that first traverse the structure's by its depth.

DFS are usually implemented with LIFO structures such as stacks, so discovered nodes are tracked. Differently from tree, DFS needs to have a structure to mark the visited nodes when traversing graphs (otherwise they might be stuck in an infinite loop). The runtime is $\mathcal{O}$(number of reachable

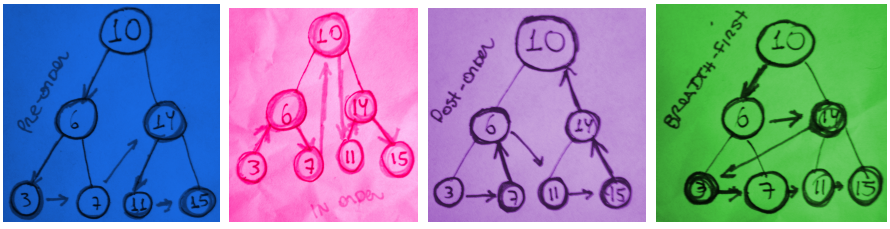


Figure 14.1: Binary tree traversals: preorder, inorder, postorder, and breath-first search.

nodes + total number of outgoing edges from these nodes) = $\mathcal{O}(V + E)$.

DFS comes in three flavors, depending on when you print (or save) the current node's value:

Preorder: Visit a node before traversing subtrees (root → left → right):

```
def preorder(root):
   if root != 0:
      yield root.value
      preorder(root.left)
      preorder(root.right)
```

Postorder: Visit a node after traversing all subtrees (left → right → root):

```
def postorder(root):
   if root != 0:
      postorder(root.left)
      postorder(root.right)
      yield root.value
```

Inorder: Visit a node after traversing its left subtree but before the right subtree (left → root → right):

```
def inorder(root):
   if root != 0:
      inorder(root.left)
      yield root.value
      inorder(root.right)
```

14.2 Breadth-First Search

Breadth-first traversals, or *breath-first search* (BFS), are algorithms that yield the values of all nodes of a particular depth *before* going to any deeper node.

Problems that employ BFSs usually are used to find the fewest number of steps (or the shortest path) needed to reach a certain end point from the starting point.

Traditionally, BFSs are implemented using a list to store the values of the visited nodes and then a FIFO *queue* to store those nodes that have yet to be visited. The total runtime is also $\mathcal{O}(V + E)$.

An example of a BFS algorithm is the following:

```
[transversal.py]

from collections import deque

def BFT(tree):
    current = tree.root
    nodes = []
    queue = deque()
    queue.append(current)

    while queue:
        current = queue.popleft()
        nodes.append(current.item)
        if current.left:
            queue.append(current.left)
        if current.right:
            queue.append(current.right)

    return nodes
```

14.3 Additional Exercises

Ancestor in a BST

The example below finds the lowest level common ancestor of two nodes in a binary search tree from an inoder list:

```
[check_ancestor.py]

from binary_search_tree import BST, Node
from transversal import inorder

def find_ancestor(path, low_value, high_value):

    while path:
        current_value = path[0]

        if current_value < low_value:
            try:
                path = path[2:]
            except:
                return current_value

        elif current_value > high_value:
            try:
                path = path[1:]
            except:
                return current_value

        elif low_value <= current_value <= high_value:
            return current_value

if __name__ == '__main__':
    bst = BST()
    l = [10, 5, 6, 3, 8, 2, 1, 11, 9, 4]
    for i in l:
        bst.add(i)
    nodes = inorder(bst.root)
    print 'Inorder:   ', nodes
    print 'Ancestor for 3, 11:', find_ancestor(nodes, 3, 11)
```

Index

Bibliography

Websites:

[**Interactive Python**] http://interactivepython.org

[**Google Style Guide**] http://google-styleguide.code.com/svn/trunk/pyguide.html

[**The git Repository** for this book] https://github.com/mariwahl/Python-and-Algorithms-and-Data-Structures

[**Big-O Sheet**] http://bigocheatsheet.com/

Books:

[*Cracking the Coding Interview]* Gayle Laakmann McDowell, 2013

[*Learn Python The Hard Way]* Zed A. Shaw, 2010

[*The Algorithm Design Manual]* S.S. Skiena, 2008