Python Programming Journey

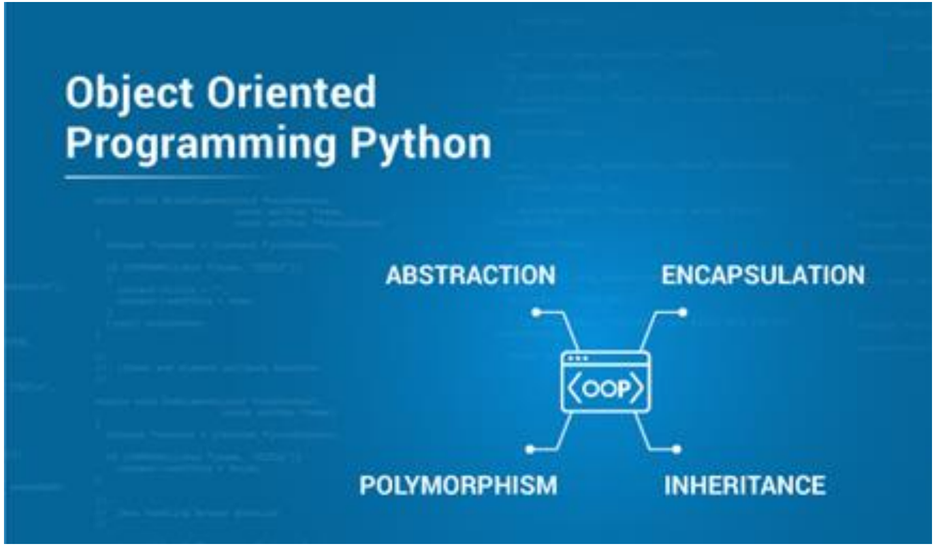
Object-Oriented Programming

LEARNING VOYAGE

Object-Oriented Programming

In this lesson, you will learn

- Writing a Class
- Displaying Objects
- Flexible Initialization
- Setters and Getters
- Inheritance
- Polymorphism
- Learning More



WRITING A CLASS

- Let’s jump right into OOP by creating a simple class to represent a person:

```
# person.py
class Person:
    """ Class to represent a person
    """
    def __init__(self):
        self.name = ''
        self.age = 0
```

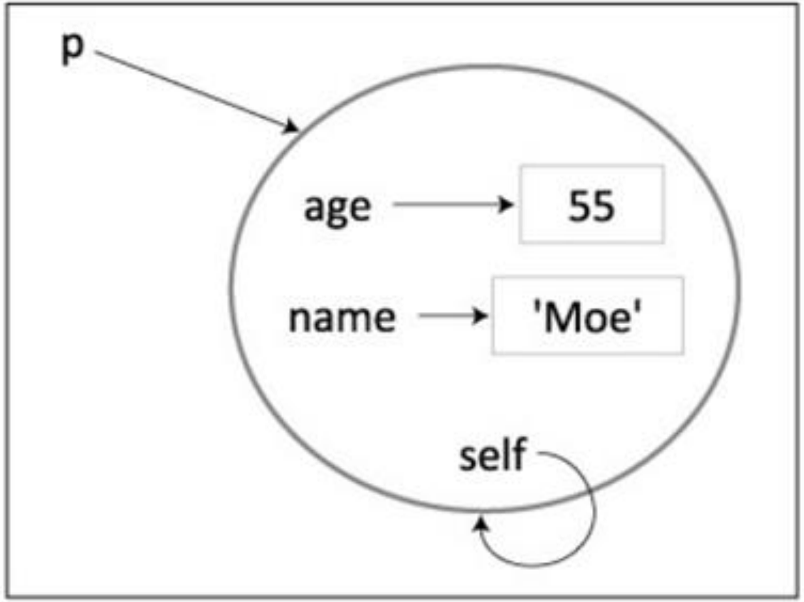
WRITING A CLASS

- We can use Person objects like this:

```
>>> p = Person()
>>> p
<__main__.Person object at 0x00AC3370>
>>> p.age
0
>>> p.name
''
>>> p.age = 55
>>> p.age
55
>>> p.name = 'Moe'
>>> p.name
'Moe'
```

The self parameter

- You’ll notice that we don’t provide any parameters for Person(), but the __init__(self) function expects an input named self.
- That’s because in OOP, self is a variable that refers to the object itself.
- This is a simple idea, but one that trips up many beginners.



DISPLAYING OBJECTS

```
# person.py
class Person:
    """ Class to represent a person
    """
    def __init__(self):
        self.name = ''
        self.age = 0
    def display(self):
        print("Person('%s', age)" % (self.name, self.age))
```

DISPLAYING OBJECTS

- The display method prints the contents of a Person object to the screen in a format useful to a programmer:

```
>>> p = Person()
>>> p.display()
Person('', 0)
>>> p.name = 'Bob'
>>> p.age = 25
>>> p.display()
Person('Bob', 25)
```

DISPLAYING OBJECTS

- For instance, the special __str__ method is used to generate a string representation of an object:

```
# person.py
class Person:
    # __init__ removed for space
    def display(self):
        print("Person('%s', age)" % (self.name, self.age))
    def __str__(self):
        return "Person('%s', age)" % (self.name, self.age)
```

DISPLAYING OBJECTS

- Now we can write code like this:

```
>>> p = Person()
>>> str(p)
"Person('', 0)"
```

DISPLAYING OBJECTS

- We can use str to simplify the display method:

```
# person.py
class Person:
    # __init__ removed for space
    def display(self):
        print(str(self))
    def __str__(self):
        return "Person('%s', age)" % (self.name, self.age)
```

DISPLAYING OBJECTS

- You can also define a special method named __repr__ that returns the “official” representation of an object.
- For example, the default representation of a Person is not very helpful:

```
>>> p = Person()
>>> p
<__main__.Person object at 0x012C3170>
```

DISPLAYING OBJECTS

- By adding a __repr__ method, we can control the string that is printed here, In most objects, it is the same as the __str__ method:

```
# person.py
class Person:
    # __init__ removed for space
    def display(self):
        print(str(self))
    def __str__(self):
        return "Person('%s', age)" % (self.name, self.age)
    def __repr__(self):
        return str(self)
```

DISPLAYING OBJECTS

- Now Person objects are easier to work with:

```
>>> p = Person()
>>> p
Person('', 0)
>>> str(p)
"Person('', 0)"
```

FLEXIBLE INITIALIZATION

- If you want to create a Person object with a particular name and age, you must currently do this:

```
>>> p = Person()
>>> p.name = 'Moe'
>>> p.age = 55
>>> p
Person('Moe', 55)
```

FLEXIBLE INITIALIZATION

- A more convenient approach is to pass the name and age to __init__ when the object is constructed. So

- let's rewrite __init__ to allow for this:

```
# person.py
class Person:
    def __init__(self, name = '',
                        age = 0):
        self.name = name
        self.age = age
```

FLEXIBLE INITIALIZATION

- Now initializing a Person is much simpler:

```
>>> p = Person('Moe', 55)
>>> p
Person('Moe', 55)
```

FLEXIBLE INITIALIZATION

- Since the parameters to __init__ have default values, you can even create an “empty” Person:

```
>>> p = Person()
>>> p
Person('', 0)
```

SETTERS AND GETTERS

- As it stands now, we can both read and write the name and age values of a Person object using dot notation

```
>>> p = Person('Moe', 55)
>>> p.age
55
>>> p.name
'Moe'
>>> p.name = 'Joe'
>>> p.name
'Joe'
>>> p
Person('Joe', 55)
```

SETTERS AND GETTERS

- First, let’s add a setter method that changes age only if a sensible value is given:

```
def set_age(self, age):
    if 0 < age <= 150:
        self.age = age
```

SETTERS AND GETTERS

- Now we can write code like this:

```
>>> p = Person('Jen', 25)
>>> p
Person('Jen', 25)
>>> p.set_age(30)
>>> p
Person('Jen', 30)
>>> p.set_age(-6)
>>> p
Person('Jen', 30)
```

Property decorators

- Property decorators combine the brevity of variables with the flexibility of functions.

- Decorators indicate that a function or method is special in some way, and here we use them to indicate setters and getters.

Property decorators

- A getter returns the value of a variable, and we indicate this using the @property decorator:

```
@property
def age(self):
    """ Returns this person's age.
    """
    return self._age
```

```python
# person.py
class Person:
    def __init__(self, name = '',
                        age = 0):
        self._name = name
        self._age = age

    @property
    def age(self):
        return self._age

    def set_age(self, age):
        if 0 < age <= 150:
            self._age = age

    def display(self):
        print(self)

    def __str__(self):
        return "Person('%s', %s)" % (self._name, self._age)

    def __repr__(self):
       return str(self)
```

LEARNING VOYAGE

Property decorators

- To create an age setter, we rename the set_age method to age and decorate it with @age.setter:

```
@age.setter
def age(self, age):
    if 0 < age <= 150:
        self._age = age
```

Property decorators

```
>>> p = Person('Lia', 33)
>>> p
Person('Lia', 33)
>>> p.age = 55
>>> p.age
55
>>> p.age = -4
>>> p.age
55
```

- With these changes, we can now write code like this:

Private variables

- It's still possible to access self._age directly:

```
>>> p._age = -44
>>> p
Person('Lia', -44)
```

Private variables

- To access self.__age directly, you now have to put _Person on the front, like this:

```
>>> p._Person__age = -44
>>> p
Person('Lia', -44)
```

INHERITANCE

- Inheritance is a mechanism for reusing classes.
- Essentially, inheritance allows you to create a brand new class by adding extra variables and methods to a copy of an existing class.

```python
# players.py
class Player:
    def __init__(self, name):
        self._name = name
        self._score = 0
    def reset_score(self):
        self._score = 0
    def incr_score(self):
        self._score = self._score + 1
    def get_name(self):
        return self._name
    def __str__(self):
        return "name = '%s', score = %s" % (self._name, self._score)
    def __repr__(self):
        return 'Player(%s)' % str(self)
```

INHERITANCE

- We can use Player objects this way:

```
>>> p = Player('Moe')
>>> p
Player(name = 'Moe', score = 0)
>>> p.incr_score()
>>> p
Player(name = 'Moe', score = 1)
>>> p.reset_score()
>>> p
Player(name = 'Moe', score = 0)
```

INHERITANCE

- We can define the Human class to inherit all the variables and methods from the Player class so that we don't have to rewrite them:

```
class Human(Player):
    pass
```

INHERITANCE

```
>>> h = Human('Jerry')
>>> h
Player(name = 'Jerry', score = 0)
>>> h.incr_score()
>>> h
Player(name = 'Jerry', score = 1)
>>> h.reset_score()
>>> h
Player(name = 'Jerry', score = 0)
```

Overriding methods

- One small wart is that the string representation of h says Player when it would be more accurate for it to say Human.
- We can fix that by giving Human its own __repr__ method:

```
class Human(Player):
    def __repr__(self):
        return 'Human(%s)' % str(self)
```

Overriding methods

- Now we get this:

```
>>> h = Human('Jerry')
>>> h
Human(name = 'Jerry', score = 0)
```

Overriding methods

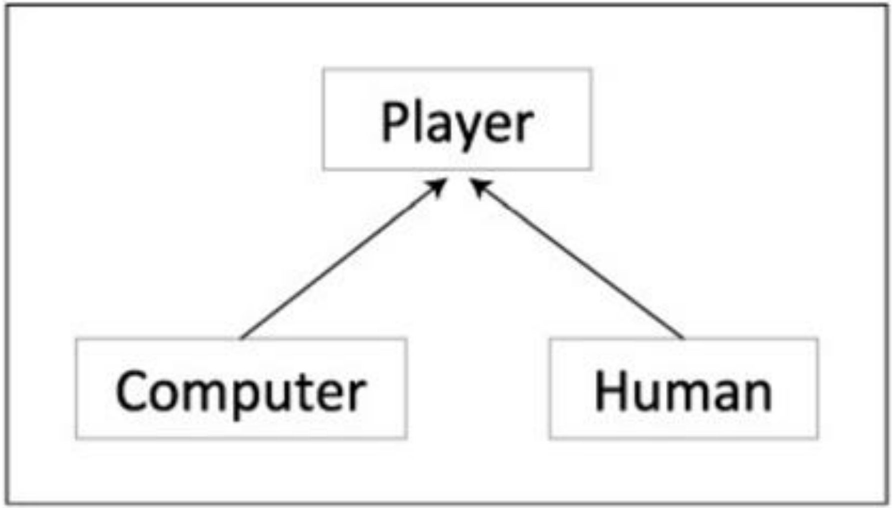
- Now it's easy to write a similar Computer class to represent computer moves:

```
class Computer(Player):
    def __repr__(self):
        return Computer(%s)' % str(self)
```

Overriding methods

- These three classes form a small class hierarchy, as shown in the class diagram.

- The Player class is called the base class, and the other two classes are derived, or extended, classes.



POLYMORPHISM

- To demonstrate the power of OOP, let's implement a simple game called Undercut.
- In Undercut, two players simultaneously pick an integer from 1 to 10 (inclusive).
- If a player picks a number one less than the other player

POLYMORPHISM

- Here’s a function for playing one game of Undercut:

```
def play_undercut(p1, p2):
    p1.reset_score()
    p2.reset_score()
    m1 = p1.get_move()
    m2 = p2.get_move()
    print("%s move: %s" % (p1.get_name(), m1))
    print("%s move: %s" % (p2.get_name(), m2))
    if m1 == m2 - 1:
        p1.incr_score()
        return p1, p2, '%s wins!' % p1.get_name()
    elif m2 == m1 - 1:
        p2.incr_score()
        return p1, p2, '%s wins!' % p2.get_name()
    else:
        return p1, p2, 'draw: no winner'
```

Implementing the move functions

- Even though moves in Undercut are just numbers from 1 to 10, humans and computers determine their moves in very different ways.

- Human players enter a number from 1 to 10 at the keyboard, whereas computer players use a function to generate their moves.

```
class Human(Player):
    def __repr__(self):
        return 'Human(%s)' % str(self)

    def get_move(self):
        while True:
            try:
                n = int(input('%s move (1 - 10): ' % self.get_name()))
                if 1 <= n <= 10:
                    return n
                else:
                    print('Oops!')
            except:
                print(Oops!')
```

Implementing the move functions

```
import random


class Computer(Player):
    def __repr__(self):
        return 'Computer(%s)' % str(self)

    def get_move(self):
        return random.randint(1, 10)
```

Playing Undercut

- With all the pieces in place, we can now start playing Undercut.
- Let’s try a game between a human and a computer:

```
>>> c = Computer('Hal Bot')
>>> h = Human('Lia')
>>> play_undercut(c, h)
Lia move (1 - 10): 7
Hal Bot move: 10
Lia move: 7
(Computer(name = 'Hal Bot', score = 0), Human(name = 'Lia', score = 0), '
```

Playing Undercut

- It’s possible to pass two computer players to play_undercut:

```
>>> c1 = Computer('Hal Bot')
>>> c2 = Computer('MCP Bot')
>>> play_undercut(c1, c2)
Hal Bot move: 8
MCP Bot move: 7
(Computer(name = 'Hal Bot', score = 0), Computer(name = 'MCP Bot', sco
```

Playing Undercut

- We can also pass in two human players:

```
>>> h1 = Human('Bea')
>>> h2 = Human('Dee')
>>> play_undercut(h1, h2)
Bea move (1 - 10): 5
Dee move (1 - 10): 4
Bea move: 5
Dee move: 4
(Human(name = 'Bea', score = 0), Human(name = 'Dee', score = 1), 'Dee w
```

LEARNING MORE

- This lesson introduced a few of the essentials of OOP.
- Python has many more OOP features you can learn about by reading the online documentation.
- Creating good object-oriented designs is a major topic.
- Using objects well is much harder than merely using them.

Case Study Text Statistics

Case Study Text Statistics

In this lesson, you will learn

- Problem Description
- Keeping the Letters We Want
- Testing the Code on a Large Data File
- Finding the Most Frequent Words
- Converting a String to a Frequency Dictionary
- Putting It All Together
- Exercises
- The Final Program

PROBLEM DESCRIPTION

- When asked to write a program that solves some non-trivial problem, beginning programmers often don’t know where to start.

- At a high level at least, the answer is simple: You start writing a big program by first understanding the problem you want to solve.

PROBLEM DESCRIPTION

Let’s look at an example using a short piece of text:

- A long time ago, in a galaxy far, far away ...

We can see that it contains:

- One line of text. We assume that the return-line character, \n, is used to indicate the end of a line, and that every text file (that is not empty!) is at least one line long.

PROBLEM DESCRIPTION

- A useful thing to do in Python is to play with examples in the interpreter.

For example:

```
>>> s = 'A long time ago, in a galaxy far, far away ...'
>>> len(s)
46
>>> s.split()
['A', 'long', 'time', 'ago,', 'in', 'a', 'galaxy', 'far,', 'far', 'away', '...']
```

PROBLEM DESCRIPTION

We will ignore non-letters (e.g., digits and punctuation), and convert uppercase letters to lowercase. So our sentence becomes this:

- Original: A long time ago, in a galaxy far, far away ...
- Modified: a long time ago in a galaxy far away

PROBLEM DESCRIPTION

- Splitting the modified sentence into words now gives more accurate results:

```
>>> t = 'a long time ago in a galaxy far far away'
>>> t.split()
['a', 'long', 'time', 'ago', 'in', 'a', 'galaxy', 'far', 'far', 'away']
>>> len(t.split())
10
```

PROBLEM DESCRIPTION

- We can count the number of unique words by converting the list to a set (recall that a set never stores duplicates):

```
>>> set(t.split())
{'a', 'ago', 'far', 'away', 'time', 'long', 'in', 'galaxy'}
>>> len(set(t.split()))
8
```

KEEPING THE LETTERS, WE WANT

- Next, let's think about how to automatically convert a string to the format we want.

- Converting a string to lowercase is easy:

```
>>> s = "I'd like a copy!"
>>> s.lower()
"i'd like a copy!"
```

KEEPING THE LETTERS WE WANT

- Getting rid of characters, we don’t want is a bit trickier. One way to do it is to use the string replace function to replace individual characters with nothing; for example:

```
>>> s = "I'd like a copy!"
>>> s.replace('!', '')
"I'd like a copy"
```

- A better approach is to keep the letters we want.

For example:

```
# Set of all characters to keep
keep = {'a', 'b', 'c', 'd', 'e',
        'f', 'g', 'h', 'i', 'j',
        'k', 'l', 'm', 'n', 'o',
        'p', 'q', 'r', 's', 't',
        'u', 'v', 'w', 'x', 'y',
        'z',
        ' ', '-', "'"}

def normalize(s):
    """Convert s to a normalized string.
    """
    result = ''
    for c in s.lower():
        if c in keep:
            result += c
    return result
```

TESTING THE CODE ON A LARGE DATA FILE

- We've written only a small amount of code, but it is enough to do some useful experiments.
- In the examples that follow, we'll use a file called bill.txt.
- It is a 5.4 megabyte text file containing the complete works of Shakespeare (which are free on the Project Gutenberg site, www.gutenberg.org).

TESTING THE CODE ON A LARGE DATA FILE

- One way to process a text file is to read the entire thing into memory as a string.
- Let's try this by hand in the interpreter:

```
>>> bill = open('bill.txt', 'r').read()
>>> len(bill)
5465395
>>> bill.count('\n')
124796
>>> len(bill.split())
904087
>>> len(normalize(bill).split())
897610
```

- Now let’s automate this by putting all the code in a function:

```
def file_stats(fname):
    """Print statistics for the given
    file.
    """
    s = open(fname, 'r').read()
    num_chars = len(s)
    num_lines = s.count('\n')
    num_words = len(normalize(s).split())

    print("The file '%s' has: " % fname)
    print("   %s characters" % num_chars)
    print("   %s lines" % num_lines)
    print("   %s words" % num_words)
```

TESTING THE CODE ON A LARGE DATA FILE

- Calling file_stats prints this:

```
>>> file_stats('bill.txt')
The file 'bill.txt' has:
    5465395 characters
    124796 lines
    897610 words
```

FINDING THE MOST FREQUENT WORDS

- Let’s consider the problem of finding the most frequently occurring words in a text file.
- The basic idea will be to use a dictionary whose keys are words and whose values are the counts of the words in the file.
- For example, consider our original example text (in normalized form):

a long time ago in a galaxy far away

FINDING THE MOST FREQUENT WORDS

We can make a count of all the words like this:

a: 2

long: 1

time: 1

ago: 1

in: 1

galaxy: 1

far: 2

away: 1

FINDING THE MOST FREQUENT WORDS

```
d = {
        'a': 2,
     'long': 1,
     'time': 1,
      'ago': 1,
       'in': 1,
   'galaxy': 1,
      'far': 2,
     'away': 1
    }
```

- If we convert this to a Python dictionary, it looks like this:

```
lst = []
for k in d:
    pair = (d[k], k)
    lst.append(pair)
#
# [(2, 'a'), (1, 'ago'),
#  (1, 'galaxy'), (1, 'time'),
#  (2, 'far'), ...]
lst.sort()
#
# [(1, 'ago'), (1, 'away'),
#  (1, 'galaxy'), (1, 'in'),
#  (1, 'long'), ...]
lst.reverse()
#
# [(2, 'far'), (2, 'a'),
#  (1, 'time'), (1, 'long'),
#  (1, 'in'), ...]
```

FINDING THE MOST FREQUENT WORDS

- With lst ordered from most frequent word to least frequent word, we can use slicing to access, say, the top three most frequent words on the list:

```
print(lst[:3])
#
# [(2, 'far'), (2, 'a'),
#  (1, 'time')]
```

FINDING THE MOST FREQUENT WORDS

Or, if we want neater formatting, we can do this:

```
for count, word in lst:
    print('%4s %s' % (count, word))
```

FINDING THE MOST FREQUENT WORDS

- Which prints:

2 far
2 a
1 time
1 long
1 in
1 galaxy
1 away
1 ago

CONVERTING ASTRING TO A FREQUENCY DICTIONARY

```
def make_freq_dict(s):
    """Returns a dictionary whose keys
        are the words of s, and whose
        values are the counts of those
        words.
    """
    s = normalize(s)
    words = s.split()
    d = {}
    for w in words:
        if w in d:    # seen w before?
            d[w] += 1
        else:
            d[w] = 1
    return d
```

PUTTING IT ALL TOGETHER

```
def print_file_stats(fname):
    """Print statistics for the given file.
    """

    s = open(fname, 'r').read()
    num_chars = len(s)           # count characters before normalizing s
    num_lines = s.count('\n')      # count lines before normalizing s
```

```
d = make_freq_dict(s)
num_words = sum(d[w] for w in d)  # count number of words in s
# create list of (count, pair) words ordered from
# most frequent to least frequent
lst = [(d[w], w) for w in d]
lst.sort()
lst.reverse()
# print the results to the screen
print("The file '%s' has: " % fname)
print("   %s characters" % num_chars)
print("   %s lines" % num_lines)
print("   %s words" % num_words)
print("\nThe top 10 most frequent words are:")
i = 1   # i is the number of the list item
for count, word in lst[:10]:
    print('%2s. %4s %s' % (i, count, word))
    i += 1
```

The file 'bill.txt' has:

5465395 characters

124796 lines

897610 words

The top 10 most frequent words are:

1. 27568 the
2. 26705 and
3. 20115 i
4. 19211 to
5. 18263 of
6. 14391 a
7. 13606 you
8. 12460 my
9. 11107 that
10. 11001 in

EXERCISES

1. Modify print_file_stats so that it also prints the total number of unique words in the file.

1. Modify print_file_stats so that it prints the average length of the words in the file

THE FINAL PROGRAM

```
# wordstats.py
# Set of all allowable characters.
keep = {'a', 'b', 'c', 'd', 'e',
        'f', 'g', 'h', 'i', 'j',
        'k', 'l', 'm', 'n', 'o',
        'p', 'q', 'r', 's', 't',
        'u', 'v', 'w', 'x', 'y',
        'z',
        ' ', '-', "'"}

def normalize(s):
    """Convert s to a normalized string.
    """
    result = ''
    for c in s.lower():
        if c in keep:
            result += c
    return result

def make_freq_dict(s):
    """Returns a dictionary whose keys are the words of s, and whose v
```

```
    are the counts of those words.
    """
    s = normalize(s)
    words = s.split()
    d = {}
    for w in words:
        if w in d:    # add 1 to its count if w has been seen before
            d[w] += 1
        else:
            d[w] = 1    # initialize to 1 if this is the first time w has been seen
    return d

def print_file_stats(fname):
    """Print statistics for the given file.
    """
    s = open(fname, 'r').read()

    num_chars = len(s)              # count characters before normalizing s
    num_lines = s.count('\n')       # count lines before normalizing s

    d = make_freq_dict(s)
    num_words = sum(d[w] for w in d)  # count number of words in s
```

```
# create list of (count, pair) words ordered from
# most frequent to least frequent
lst = [(d[w], w) for w in d]
lst.sort()
lst.reverse()

# print the results to the screen
print("The file '%s' has: " % fname)
print("   %s characters" % num_chars)
print("   %s lines" % num_lines)
print("   %s words" % num_words)

print("\nThe top 10 most frequent words are:")
i = 1   # i is the number of the list item
for count, word in lst[:10]:
    print('%2s. %4s %s' % (i, count, word))
```

THE FINAL PROGRAM

```
        i += 1

def main():
    print_file_stats('bill.txt')

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

Demo