Writing Structured Programs

Behrang QasemiZadeh

Objectives

- With a focus on NLP applications, we discuss:
  - Writing well-structured and readable codes.
  - Reviewing fundamental building blocks of Python programming:
    - Functions;
    - Control structures such as loops;
    - Assignments;
    - Programming constructs.
  - Gaining knowledge about shortcomings of Python.

Assignment

- An assignment statement sets the value stored by a variable name.
  - The assignment operator is the equal sign “=”.
  - The **name** of the variable is always on the left side of the equals sign, and the **value** of the variable on the right side of the equals sign.

```
>>> foo = 'Monty'
```

Assignment

- An assignment statement sets the value stored by a variable name.
  - The assignment operator is the equal sign “=”.
  - The **name** of the variable is always on the left side of the equals sign, and the **value** of the variable on the right side of the equals sign.

```
>>> foo = 'Monty'
```

  - A variable, object fields, and entries in collections, etc. are just **references**.
  - **Values** are stored else where and **referenced** by variables.
  - Multiple **references** can refer to the same value.
  - In simplest terms, a variable is just a box that you can put stuff in.

Assignment Quiz: basic data type

What is the output for bar?!

```
>>> foo = 'Monty'
>>> bar = foo
>>> foo = 'Python'
>>> bar
```

Assignment Quiz: basic data type

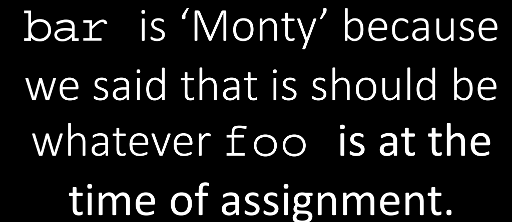
What is the output for bar?!

```
>>> foo = 'Monty'
>>> bar = foo
>>> foo = 'Python'
>>> bar
'Monty'
```

Assignment Quiz: basic data type

What is the output for bar?!

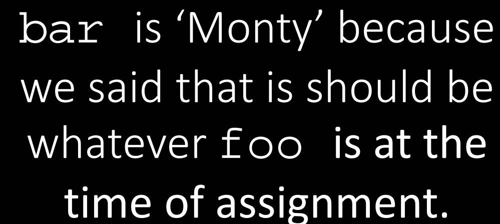
```
>>> foo = 'Monty'
>>> bar = foo
>>> foo = 'Python'
>>> bar
'Monty'
```

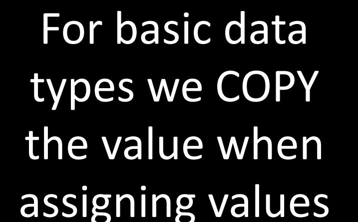


Assignment Quiz: basic data type

What is the output for bar?!

```
>>> foo = 'Monty'
>>> bar = foo
>>> foo = 'Python'
>>> bar
'Monty'
```





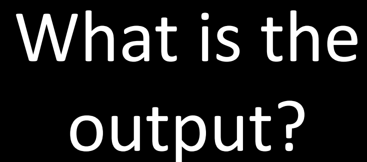
Assignment: lists and dictionaries

- The “value” of a structured object such as a list or a dictionary is actually just a **reference** to the object.

Assignment: lists and dictionaries

- The “value” of a structured object such as a list or a dictionary is actually just a **reference** to the object.

```
>>> foo = ['Monty', 'Python']
>>> bar = foo
>>> foo[1] = 'Bodkin'
>>> bar
```



Assignment: lists and dictionaries

- The “value” of a structured object such as a list or a dictionary is actually just a **reference** to the object.

```
>>> foo = ['Monty', 'Python']
>>> bar = foo
>>> foo[1] = 'Bodkin'
>>> bar
['Monty', 'Bodkin']
```

Assignment: lists and dictionaries

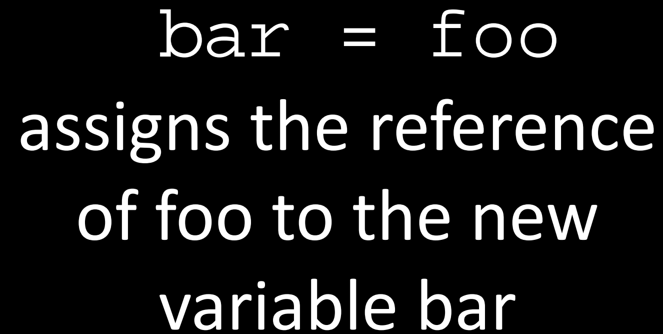
- The "value" of a structured object such as a list or a dictionary is actually just a **reference** to the object.

```
>>> foo = ['Monty', 'Python']
>>> bar = foo
>>> foo[1] = 'Bodkin'
>>> bar
['Monty', 'Bodkin']
```

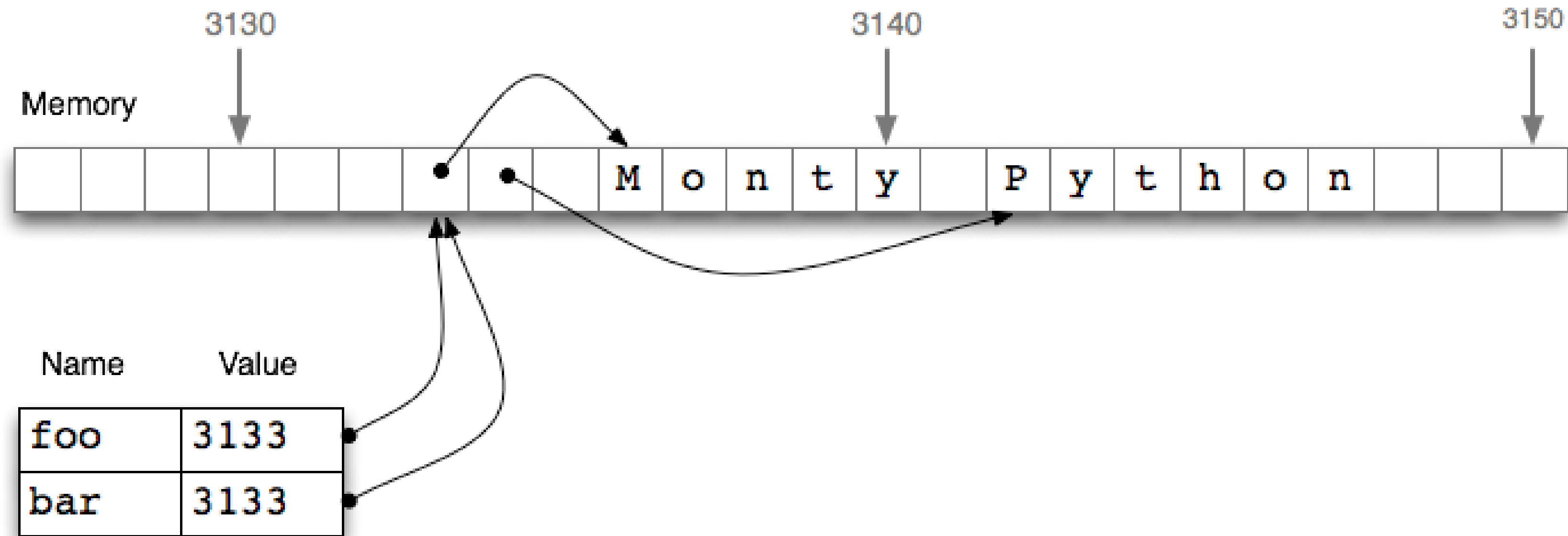
Assignment: lists and dictionaries

- The "value" of a structured object such as a list or a dictionary is actually just a **reference** to the object.

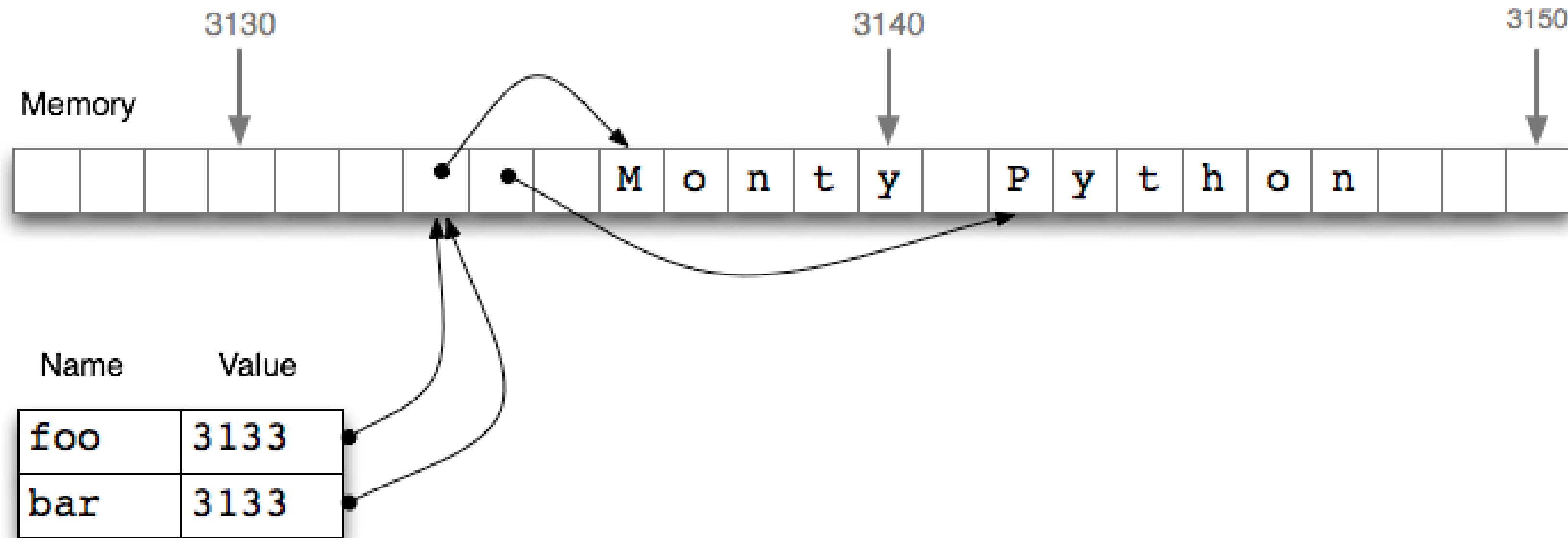
```
>>> foo = ['Monty', 'Pytho
>>> bar = foo
>>> foo[1] = 'Bodkin'
>>> bar
['Monty', 'Bodkin']
```



Assignment: lists and dictionaries



Assignment: lists and dictionaries



- Read more on data types: http://en.wikibooks.org/wiki/Python_Programming/Data_Types

Quiz

- What is the output?

```
>>> empty = []
>>> nested = [empty, empty, empty]
>>> nested[1].append('Python')
>>> nested
```

```
>>> nested = [[]] * 3
>>> nested[1].append('Python')
>>> nested[1] = ['Monty']
>>> nested
```

Quiz

- What is the output?

```
>>> empty = []
>>> nested = [empty, empty, empty]
>>> nested[1].append('Python')
>>> nested

[['Python'], ['Python'], ['Python']]
```

```
>>> nested = [[]] * 3
>>> nested[1].append('Python')
>>> nested[1] = ['Monty']
>>> nested

[['Python'], ['Monty'], ['Python']]
```

Quiz

- What is the output?

```
>>> empty = []
>>> nested = [empty, empty, empty]
>>> nested[1].append('Python')
>>> nested

[['Python'], ['Python'], ['Python']]
```

```
>>> nested = [[]] * 3
>>> nested[1].append('Python')
>>> nested[1] = ['Monty']
>>> nested

[['Python'], ['Monty'], ['Python']]
```

Quiz

- What is the output?

```
>>> empty = []
>>> nested = [empty, empty, empty]
>>> nested[1].append('Python')
>>> nested
```

```
[['Python'], ['Python'
```

```
>>> nested = [[]] * 3
>>> nested[1].append('Python')
>>> nested[1] = ['Monty']
>>> nested
```

```
nty'], ['Python']]
```

`id()` function Return the identity of an object. Try function `id()`, e.g. type in `id(nested[1])`

Assignment

- Assignment statements in Python do not copy objects, they create bindings between a target and an object.
- To copy items from a list called `foo` to a new list called `bar` you can use `bar = foo[:]`
  - Compare `bar = foo[:]` and `bar = foo`
  - What if foo and bar contains lists?
- Shallow and deep copy operations can be performed using `copy`
  - A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original.
  - A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.

Equality and identity

- It is sometimes necessary to compare two values for equality.
- The `is` operator tests for object **identity (remember `id()?!`)**:
  - in the previous slide, what is the output of `nested[0] is nested[1]`?
- Identity (`is`) implies equality (`==`) but the reverse is not true:
  - Two distinct objects can have the same value.
- You use `==` when comparing values and `is` when comparing identities.

Conditionals

```
>>> mixed = ['cat', '', ['dog'], []]
>>> for element in mixed:
     if element:
          print element
cat
['dog']
>>>
```

- In the condition part of an if statement, a non-empty string or list is evaluated as true, while an empty string or list evaluates as false.

Conditionals

- Be informed of the difference of `elif` and a number of consecutive `if` statements:
  - The satisfaction of `if` statements, terminates the rest of comparisons in the conditional construct.

```
>>> animals = ['cat', 'dog']
>>> if 'cat' in animals:
     print(1)
        elif 'dog' in animals:
     print(2)
 1
```

Conditionals

- The functions `all()` and `any()` can be applied to a list (or other sequence) to check condition for all or ant items:
  - They can be used for writing more natural and concise codes!

```
>>> sent = ['No', 'good', 'fish', 'goes', 'anywhere',
'without', 'a', 'porpoise', '.']
>>> all(len(w) > 4 for w in sent)
False
>>> any(len(w) > 4 for w in sent)
True
```

Sequences: String, List, Tuple

- Sequence data types can be sliced, indexed and they have length:

```
>>> t = "passau", "Innstr", 94032
>>> t[0]
'passau'
>>> t[1:]
('Innstr', 94032)
>>> len(t)
3
>>>
```

Sequences: String, List, Tuple

- Iterations over a sequence is common:

Python Expression	Comment
`for item in s`	iterate over the items of s
`for item in sorted(s)`	iterate over the items of s in order
`for item in set(s)`	iterate over unique elements of s
`for item in reversed(s)`	iterate over elements of s in reverse
`for item in set(s).difference(t)`	iterate over elements of s not in t

Sequences: String, List, Tuple

- Iterations over a sequence is common:

Python Expression	Comment
`for item in s`	iterate over the items of s
`for item in sorted(s)`	iterate over [illegible]
`for item in set(s)`	[illegible]
`for item in reversed(s)`	
`for item in set(s).differ`[illegible]	

You can combine the sequence functions in a variety of way:
`reverse(sorted(set(s)))`

Sequences: String, List, Tuple

- One sequence type can be converted into another one:
  - `tuple(s)` convers any kind of sequence to tuple;
  - `list(s)` converts any kind of sequence to list;
  - `join()` convert a sequence to string.
- Other objects (such as the `FreqDist`)can be converted to a sequence(`e.g. FreqDist` to list).

Sequences: String, List, Tuple

- One sequence type can be converted into another one:
  - `tuple(s)` convers any kind of sequence to tuple;
  - `list(s)` converts any kind of sequence to list;
  - `join()` convert a sequence to string.

```
>>> raw = 'Red lorry, yellow lorry, red lorry, yellow lorry.'
>>> text = nltk.word_tokenize(raw)
>>> fdist = nltk.FreqDist(text)
>>> sorted(fdist)
[',', '.', 'Red', 'lorry', 'red', 'yellow']
>>> for key in fdist:
      print(key + ':', fdist[key], end='; ')
lorry: 4; red: 1; .: 1; ,: 3; Red: 1; yellow: 2
```

Sequences: String, List, Tuple

- There are functions that modify the **structure** of a sequence and which can be handy for language processing.
- **`zip()`** takes the items of two or more sequences and "zips" them together into a single list of tuples.

Sequences: String, List, Tuple

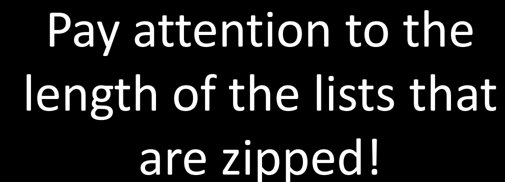
- There are functions that modify the **structure** of a sequence and which can be handy for language processing.
- `zip()` takes the items of two or more sequences and "zips" them

```
>>> words = ['I', 'turned', 'off', 'the', 'spectroroute']
>>> tags = ['noun', 'verb', 'prep', 'det', 'noun']
>>> zip(words, tags)
<zip object at ...>
>>> list(zip(words, tags))
[('I', 'noun'), ('turned', 'verb'), ('off', 'prep'), ('the', 'det'), ('spectroroute', 'noun')]
>>> list(enumerate(words))
[(0, 'I'), (1, 'turned'), (2, 'off'), (3, 'the'), (4, 'spectroroute')]
```

Sequences: String, List, Tuple

- There are functions that modify the **structure** of a sequence and which can be handy for language processing.
- `zip()` takes the items of two or more sequences and "zips" them

```
>>> words = ['I', 'turned', 'off', 'the', 'spectroroute']
>>> tags = ['noun', 'verb', 'prep', 'det', 'noun']
>>> zip(words, tags)
<zip object at ...>
>>> list(zip(words, tags))
[('I', 'noun'), ('turned', 'verb'), ('off', 'prep'),                'noun')]
>>> list(enumerate(words))
[(0, 'I'), (1, 'turned'), (2, 'off'), (3, 'the                ]
```



Sequences: String, List, Tuple

- For some NLP tasks it is necessary to cut up a sequence into two or more parts.
  - Example: 80% of a corpus for train and 20% for test!
  - This can be achieved by slicing

```
>>> text = nltk.corpus.nps_chat.words()
>>> cut = int(0.9 * len(text))
>>> training_data, test_data = text[:cut], text[cut:]
>>> text == training_data + test_data
True
>>> len(training_data) / len(test_data)
9.0
```

Quiz

What is the output of the following code snippet? Can you explain each line of this code?

```
>>> words = 'I turned off the spectroroute'.split()
>>> wordlens = [(len(word), word) for word in words]
>>> wordlens.sort()
>>> ' '.join(w for (_, w) in wordlens)
```

Quiz

What is the output of the following code snippet? Can you explain each line of this code?

```
>>> words = 'I turned off the spectroroute'.split()
>>> wordlens = [(len(word), word) for word in words]
>>> wordlens.sort()
>>> ' '.join(w for (_, w) in wordlens)
'I off the turned spectroroute'
```

Sequences: When to Use What!

- String: in the beginning and the end:
  - Typical when reading in some text and producing output for us to read.
- Lists and tuples are used in the middle, but for different purposes.

Sequences: When to Use What!

- String: in the beginning and the end:
  - Typical when reading in some text and producing output for us to read.
- A list is typically a sequence of objects with the following condition:
  - All objects have the **same type*;***
  - The **length** of list is **not fixed** and can be changed at any time;
  - We often use lists to hold **sequences of words.**

Sequences: When to Use What!

- String: in the beginning and the end:
  - Typical when reading in some text and producing output for us to read.
- A list is typically a sequence of objects with the following condition:
  - All objects have the **same type*;***
  - The **length** of list is **not fixed** and can be changed at any time;
  - We often use lists to hold **sequences of words.**
- A tuple, however, is typically a collection of objects:
  - Objects have **different types;**
  - The **length** of tuple is **fixed**.
  - We often use tuples to represent **records** (a collection of various **fields** about some entity).

Sequences: When to Use What!

- String: in the beginning and the end:
  - Typical when reading in some text and producing output for us to read.
- A list is typically a sequence of objects with the following condition:
  - All objects have the **same type*;***
  - The **length** of list is **not fixed** and can be changed at any time;
  - We often use lists to hold **sequences of words.**
- A tuple, however, is typically a collection of objects:
  - Objects have **different types;**
  - The **length** of tuple is **fixed**.
  - We often use tuples to represent **records** (a collection of various **fields** about some entity).

Sequences: mutable vs immutable

- Strings are immutable.
  - You cannot sort characters in a string;
- Tuples are immutable.
  - You cannot sort the elements of a tuple;
- Lists are mutable.
  - But, you can sort a list!

Generator Expressions

- Compare the following code snippets:

```
>>> max([w.lower() for w in word_tokenize(text)])
'word'

>>> max(w.lower() for w in word_tokenize(text))
'word'
```

Generator Expressions

- Compare the following code snippets:

```
>>> max([w.lower() for w in word_tokenize(text)])
'word'

>>> max(w.lower() for w in word_tokenize(text))
'word'
```

Generator Expressions

- Compare the following code snippets:

```
>>> max([w.lower() for w in word_tokenize(text)])
'word'

>>> max(w.lower() for w in word_tokenize(text))
'word'
```

Generator Expressions

- Compare the following code snippets:

```
>>> max([w.lower() for w in word_tokenize(text)])
```

The list object must be allocated before the value of `max( )` is computed.

```
>>> max(w.lower() for w in word_tokenize(text))
'word'
```

Generator Expressions

- Compare the following code snippets:

```
>>> max([w.lower() for w in word_tokenize(text)])
```

The list object must be allocated before the value of `max( )` is computed.

```
>>> max(w.lower() for w in word_tokenize(text))
'word'
```

The data is streamed to the calling function (`max( )` does not need to store and access the whole list)

Generator Expressions

- Compare the following code snippets:

```
>>> max([w.lower() for w in word_tokenize(text)])
```

The list object must be allocated before the value of `max()` is computed.

```
>>> max(w.lower() for w in word_tokenize(text))
'word'
```

The data is streamed to the calling function (`max()` does not need to store and access the whole list)

Generator Expressions

- Compare the following code snippets:

```
>>> max([w.lower() for w in word_tokenize(text)])
```

The list object must be allocated before the value of `max( )` is computed.

```
>>> max(w.lower() for w in word_tokenize(text))
'word'
```

The data is streamed to the calling function (`max( )` does not need to store and access the whole list)

Let’s have a little break!

Python Style!

- No, we are not talking about poor pythons skin!
  - **pythons** are hunted in Indonesia and Malaysia, and species are threatened.
- We are talking about more civilized choices:
  - variable names;
  - Spacing;
  - comments, etc.

Python Style!

- A style guide for Python code can be found at https://www.python.org/dev/peps/pep-0008/.
- The most important matter the style guide is consistency.
- The goal is to improve readability.
  - This is important specifically when you are working in teams.

Python Style!

- Lines should be less than 80 characters long:
  - Break a line inside parentheses, brackets, or braces;
  - Add extra parentheses;
  - And, you can always add a backslash at the end of the line that is broken.
- Remember that the indentation of blocks of code is not the matter of choice (4 space character):
  - Spaces are the preferred indentation method.
  - Most editor does the automatic indent.

Python Style!

- From https://www.python.org/dev/peps/pep-0008 see also:
  - Whitespace in Expressions and Statements
  - Naming Conventions
  - Version bookkeeping
  - Comments
  - etc.

Procedural vs Declarative Style

- Compare these two code snippets (yellow and green blocks):

```
>>> tokens = nltk.corpus.brown.words(categories='news')
>>> count = 0
>>> total = 0
>>> for token in tokens:
      count += 1
      total += len(token)
>>> total / count
4.401545438271973
```

```
>>> total = sum(len(t) for t in tokens)
>>> print(total / len(tokens))
4.401...
```

Procedural vs Declarative Style

- Compare these two code snippets (yellow and green blocks):

```
>>> tokens = nltk.corpus.brown.words(categories='news')
>>> count = 0
>>> total = 0
>>> for token in tokens:
        count += 1
        total += len(token)
>>> total / count
4.401545438271973
```

```
>>> total = sum(len(t) for t in tokens)
>>> print(total / len(tokens))
4.401...
```

Procedural vs Declarative Style

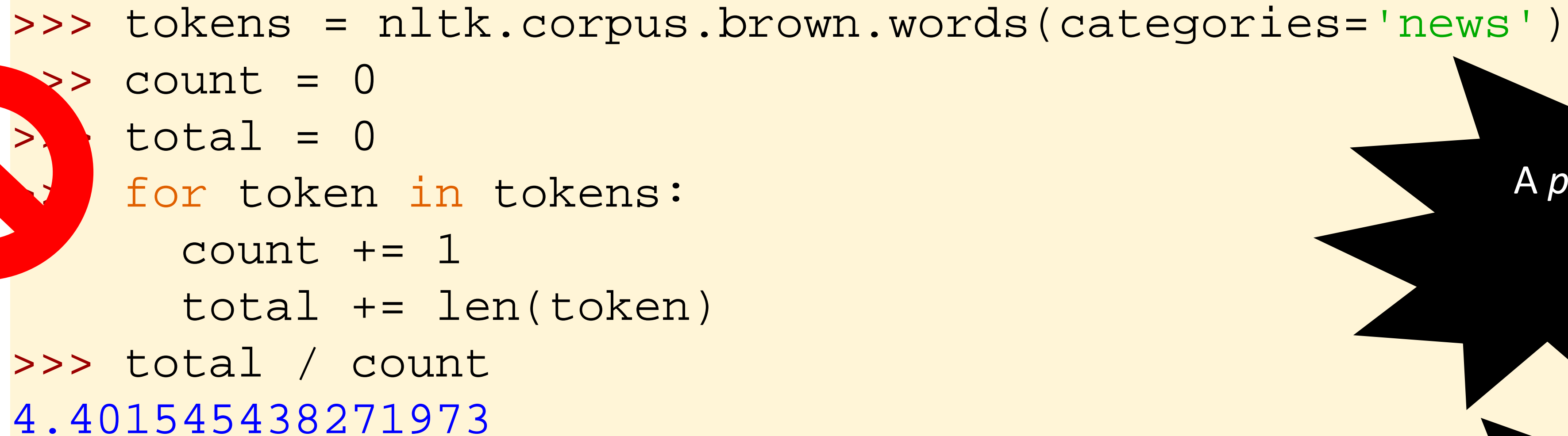
- Compare these two code snippets (yellow and green blocks):

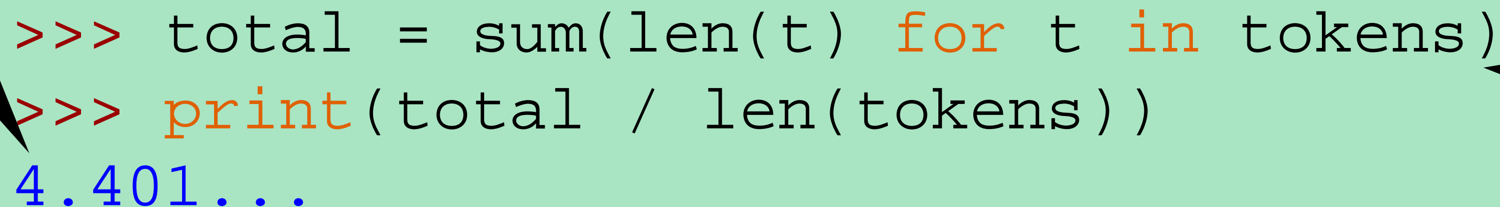
```
>>> tokens = nltk.corpus.brown.words(categories='news')
>>> count = 0
>>> total = 0
>>> for token in tokens:
      count += 1
      total += len(token)
>>> total / count
4.401545438271973
```

```
>>> total = sum(len(t) for t in tokens)
>>> print(total / len(tokens))
4.401...
```

Procedural vs Declarative Style

- Compare these two code snippets (yellow and green blocks):





READ and PRACTICE!

Structured Programming Using Functions

- Functions provide an effective way to reuse and package code.
- Using functions has the benefit of
  - Saving space in our program;
  - Improving the readability of our codes;
  - Easing code maintenance, debugging and upgrades.
- Well-structured programs usually make extensive use of functions.
- A block of code longer than 10-20 lines must be decomposed into multiple functions.

Structured Programming Using Functions

```python
import re
def get_text(file):
    """Read text from a file, normalizing whitespace and stripping HTML markup."""
    text = open(file).read()
    text = re.sub(r'<.*?>', ' ', text)
    text = re.sub('\s+', ' ', text)
    return text
```

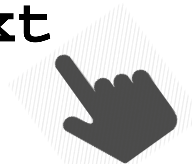
Structured Programming Using Functions

Argument (call-by-value)

Docstring (to be used for `help()`).

```
import re
def get_text(file):
    """Read text from a file, normalizing whitespace and stripping HTML markup."""
    text = open(file).read() # and we know how to add comments
    text = re.sub(r'<.*?>', ' ', text)
    text = re.sub('\s+', ' ', text)
    return text
```

body



Return statement

Variable Scope

- Function definitions create a new, **local scope** for variables.
  - The name is not visible outside the function, or in other functions.
  - You can choose variable names without being concerned about collisions with names used in your other function definitions.
- Resolving variable names (**LGB rule**):
  - The Python interpreter first tries to resolve the name with respect to the names that are local to the function.
  - If nothing is found, the interpreter checks if it is a global name within the module.
  - Finally, if that does not succeed, the interpreter checks if the name is a Python built-in.

Checking Parameter Types

- Python doesn't let us to declare the type of a variable.
  - This permits us to define functions that are flexible about the type of their arguments.
- In a defensive style of programming, we may want to check the type of arguments:
  - A naive approach would be to check the type of the argument using `if not type(X) is Y`, e.g. a string variable of type `str`.
    - Dangerous because the calling program may not detect the `None` output of the `if` statement properly.
  - Using an `assert` statement is a safer choice.
    - `If assert` fails, it will produce an error that cannot be ignored.

Checking Parameter Types

```
>>> def tag(word):
        assert isinstance(word, basestring), "argument to tag() must be a string"
        if word in ['a', 'the', 'all']:
                return 'det'
        else:
                return 'noun'
```

Checking Parameter Types

`isinstance` checks to see if the object passed in the first argument is of the type of any of the type objects passed in the second argument.

```
>>> def tag(word):
        assert isinstance(word, basestring), "argument to tag() must be a string"
        if word in ['a', 'the', 'all']:
                return 'det`
        else:
                return 'noun'
```

Documenting Functions

- For the simplest functions, a one-line **docstring** is usually adequate:
  - a triple-quoted string containing a complete sentence on a single line.
- For non-trivial functions, consider providing docsrting followed by a blank line, then a more detailed description of the functionality
- Docsrting can also include a **doctest block**, illustrating the use of the function and the expected output.
  - Look into `docutils` module.

Documenting Functions

- Docstrings should document the **type of each parameter** to the function, **and the return type**.
- Docstring can be a simple text, however, you can also use some kind of markup language for documentation.
- NLTK uses the Sphinx markup language
  - http://sphinx-doc.org/index.html
  - Sphinx markups can be converted into richly structured API documentation.
    - Output formats: HTML, LaTeX, ePub, Texinfo, manual pages, plain text
    - Extensive cross-references:
    - Hierarchical structure
    - Automatic indices
    - Code handling
    - Extensions, etc.

Advanced Features of Functions

- **Functions as Arguments**
  - Python lets us pass a function as an argument to another function.
  - This lets us abstract out the operation, and apply a different operation on the same data.

Advanced Features of Functions

- **Functions as Arguments**
  - Python lets us pass a function as an argument to another function.
  - This lets us abstract out the operation, and apply a different operation on the same data.

```
>>> sent = ['Take', 'care', 'of', 'the', 'sense’]
>>> def extract_property(prop):
        return [prop(word) for word in sent]
>>> extract_property(len)
[4, 4, 2, 3, 5]
>>> def last_letter(word):
        return word[-1]
>>> extract_property(last_letter)
['e', 'e', 'f', 'e', 'e']
```

Advanced Features of Functions

- **Functions as Arguments**
  - Python lets us pass a function as an argument to another function.
  - This lets us abstract out the operation, and apply a different operation on the same data.
  - We can also use **lambda expressions**.

Advanced Features of Functions

- **Functions as Arguments**
  - Python lets us pass a function as an argument to another function.
  - This lets us abstract out the operation, and apply a different operation on the same data.
  - We can also use **lambda expressions**.

```
>>> extract_property(lambda w: w[-1])
['e', 'e', 'f', 'e', 'e']
```

Advanced Features of Functions

- **Functions as Arguments**
  - Python lets us pass a function as an argument to another function.
  - This lets us abstract out the operation, and apply a different operation on the same data.
  - We can also use **lambda expressions**.

```
>>> sent = ['hello', 'and', 'greetings', 'friends']
>>> sorted(sent)
['and', 'friends', 'greetings', 'hello']
>>> sorted(sent, cmp)
['and', 'friends', 'greetings', 'hello']
>>> sorted(sent, lambda x, y: cmp(len(y), len(x)))
['greetings', 'friends', 'hello', 'and']
```

Advanced Features of Functions

- **Accumulative functions**
  - These functions start by initializing some storage, and iterate over input to build it up, before returning some final object:

```
def search1(substring, words):
        result = []
        for word in words:
                if substring in word:
                        result.append(word)
        return result
```

Advanced Features of Functions

- **Accumulative functions**
  - These functions start by initializing some storage, and iterate over input to build it up, before returning some final object:

```
def search1(substring, words):
    result = []
    for word in words:
        if substring in word:
            result.append(word)
    return result
```

```
def search2(substring, words):
    for word in words:
        if substring in word:
            yield word
```

Advanced Features of Functions

- **Accumulative functions**
  - These functions start by initializing some storage, and iterate over input to build it up, before returning some final object:

```
def search1(substring, words):
        result = []
        for word in words:
                if substring in word:
                        result.append(word)
        return result
```

```
def search2(substring, words):
        for word in words:
                if substring in word:
                        yield word
```

- Function `search2()` is a generator:
  - The first time `search2()` is called, it gets as far as the `yield` statement and pauses.
  - The calling program gets the first word and does any necessary processing.
  - Once the calling program is ready for another word, execution of `search2()` is continued from where it stopped, until the next time it encounters a `yield` statement.

```
def search1(substring, words):
        result = []
        for word in words:
                if substring in word:
                        result.append(word)
        return result
```

```
def search2(substring, words):
        for word in words:
                if substring in word:
                        yield word
```

Advanced Features of Functions

- **Higher order functions**
  - `filter()` applies the function to each item in the sequence contained in its second parameter, and retains only the items for which the function returns `True`.

```
>>> def is_content_word(word):
        return word.lower() not in ['a', 'of', 'the', 'and', 'will', ',', '.']
>>> sent = ['Take', 'care', 'of', 'the', 'sense', ',', 'and', 'the',
... 'sounds', 'will', 'take', 'care', 'of', 'themselves', '.']
>>> list(filter(is_content_word, sent))
['Take', 'care', 'sense', 'sounds', 'take', 'care', 'themselves']
>>> [w for w in sent if is_content_word(w)]
['Take', 'care', 'sense', 'sounds', 'take', 'care', 'themselves']
```

Advanced Features of Functions

- **Higher order functions**
  - **`filter()`** applies the function to each item in the sequence contained in its second parameter, and retains only the items for which the function returns **`True`**.
  - **`map()`** applies a function to every item in a sequence.

Advanced Features of Functions

- **Higher order functions**
  - **`filter()`** applies the function to each item in the sequence contained in its second parameter, and retains only the items for which the function returns **`True`**.
  - **`map()`** applies a function to every item in a sequence.

```
>>> lengths = map(len, nltk.corpus.brown.sents(categories='news'))
>>> sum(lengths) / len(lengths)
21.7508116158339
>>> lengths = [len(sent) for sent in nltk.corpus.brown.sents(categories='news')]
>>> sum(lengths) / len(lengths)
21.7508116158339
```

Advanced Features of Functions

- **Named Arguments**
  - When a function gets a lot of arguments, it is easy to get confused about the correct order.
  - We can refer to parameters by name, and even assign them a default value.

Advanced Features of Functions

- **Named Arguments**
  - When a function gets a lot of arguments, it is easy to get confused about the correct order.
  - We can refer to parameters by name, and even assign them a default value.

```
>>> def repeat(msg='<empty>', num=1):
      return msg * num
>>> repeat(num=3)
'<empty><empty><empty>'
>>> repeat(msg='Alice')
'Alice'
>>> repeat(num=5, msg='Alice')
'AliceAliceAliceAliceAlice'
```

Advanced Features of Functions

- **Named Arguments**
  - When a function gets a lot of arguments, it is easy to get confused about the correct order.
  - We can refer to parameters by name, and even assign them a default value.

```
>>> def repeat(msg='<empty>', num=1):
        return msg * num
>>> repeat(num=3)
'<empty><empty><empty>'
>>> repeat(msg='Alice')
'Alice'
>>> repeat(num=5, msg='Alice')
'AliceAliceAliceAliceAlice'
```

Advanced Features of Functions

- **Named Arguments**
  - When a function gets a lot of arguments, it is easy to get confused about the correct order.
  - We can refer to parameters by name, and even assign them a default value.
  - We can define a function that takes an arbitrary number of unnamed and named arguments.

Advanced Features of Functions

- **Named Arguments**
  - When a function gets a lot of arguments, it is easy to get confused about the correct order.
  - We can refer to parameters by name, and even assign them a default value.
  - We can define a function that takes an arbitrary number of unnamed and named arguments.

```
>>> def generic(*args, **kwargs):
      print(args)
      print(kwargs)
>>> generic(1, "African swallow", monty="python")
(1, 'African swallow') {'monty': 'python'}
```

Advanced Features of Functions

- **Named Arguments**
  - When a function gets a lot of arguments, it is easy to get confused about the correct order.
  - We can refer to parameters by name, and even assign them a default value.
  - We can define a function that takes an arbitrary number of unnamed and named arguments.

All the unnamed parameters

The keyword arguments

```
>>> def generic(*args, **kwargs):
        print(args)
        print(kwargs)
>>> generic(1, "African swallow", monty="python")
(1, 'African swallow') {'monty': 'python'}
```

Advanced Features of Functions

- **Named Arguments**
  - When a function gets a lot of arguments, it is easy to get confused about the correct order.
  - We can refer to parameters by name, and even assign them a default value.
  - We can define a function that tak… …mber of unnamed and named arguments.

You can use the *args in other occasions: a short hand to denote the items in a list, e.g. args[0], args[1], args[2], etc.

```
>>> def generic(*args,
      print(args)
      print(kwargs)
>>> generic(1, "African swallow", monty="python")
(1, 'African swallow') {'monty': 'python'}
```

Advanced Features of Functions

- **Named Arguments**
  - When a function gets a lot of arguments, it is easy to get confused about the correct order.
  - We can refer to parameters by name, and even assign them a default value.
  - We can define a function that takes an arbitrary number of unnamed and named arguments.
  - **Caution**:
    - Do not use mutable objects as default values of arguments.
    - If you work with files, then it is a good practice to close them afterwards.
      - Use keyword with to ask Python to take care after it automatically.

Advanced Features of Functions

- **Named Arguments**
  - When a function gets a lot of arguments, it is easy to get confused about the correct order.
  - We can refer to parameters by name, and even assign them a default value.
  - We can define a function that takes an arbitrary number of unnamed and named arguments.
  - **Caution**:
    - Do not use mutable objects as default values of arguments.
    - If you work with files, then it is a good practice to close them afterwards.
      - Use keyword with to ask Python to take care after it automatically.

```
>>> with open("lexicon.txt") as f:
        data = f.read()
```

Program Development

- Programming is a skill that comes with experience.
- Everybody can program if they practice! It is similar to speaking a new language.

Program Development

- Programming is a skill that comes with experience.
- Everybody can program if they practice! It is similar to speaking a new language.
- In order to be a good programmer, you need knowledge about:
  - Algorithm Design
  - Structured Programming
  - Knowledge of the syntax of your programming language (keywords and conditional structures, loops, etc.)
  - Test methods for trouble-shooting and debugging

Program Development

- Programming is a skill that comes with experience.
- Everybody can program if they practice! It is similar to speaking a new language.
- In order to be a good programmer, you need knowledge about:
  - Algorithm Design
  - Structured Programming
  - Knowledge of the syntax of your programming language (keywords and conditional structures, loops, etc.)
  - Test methods for trouble-shooting and debugging

Structure of a Python Module

- Module is used to bring logically-related definitions and functions together:
  - The goal is to facilitate re-use and abstraction.
  - An individual `.py` file is a Python module.
    - For example, you can group all your I/O methods.

Structure of a Python Module

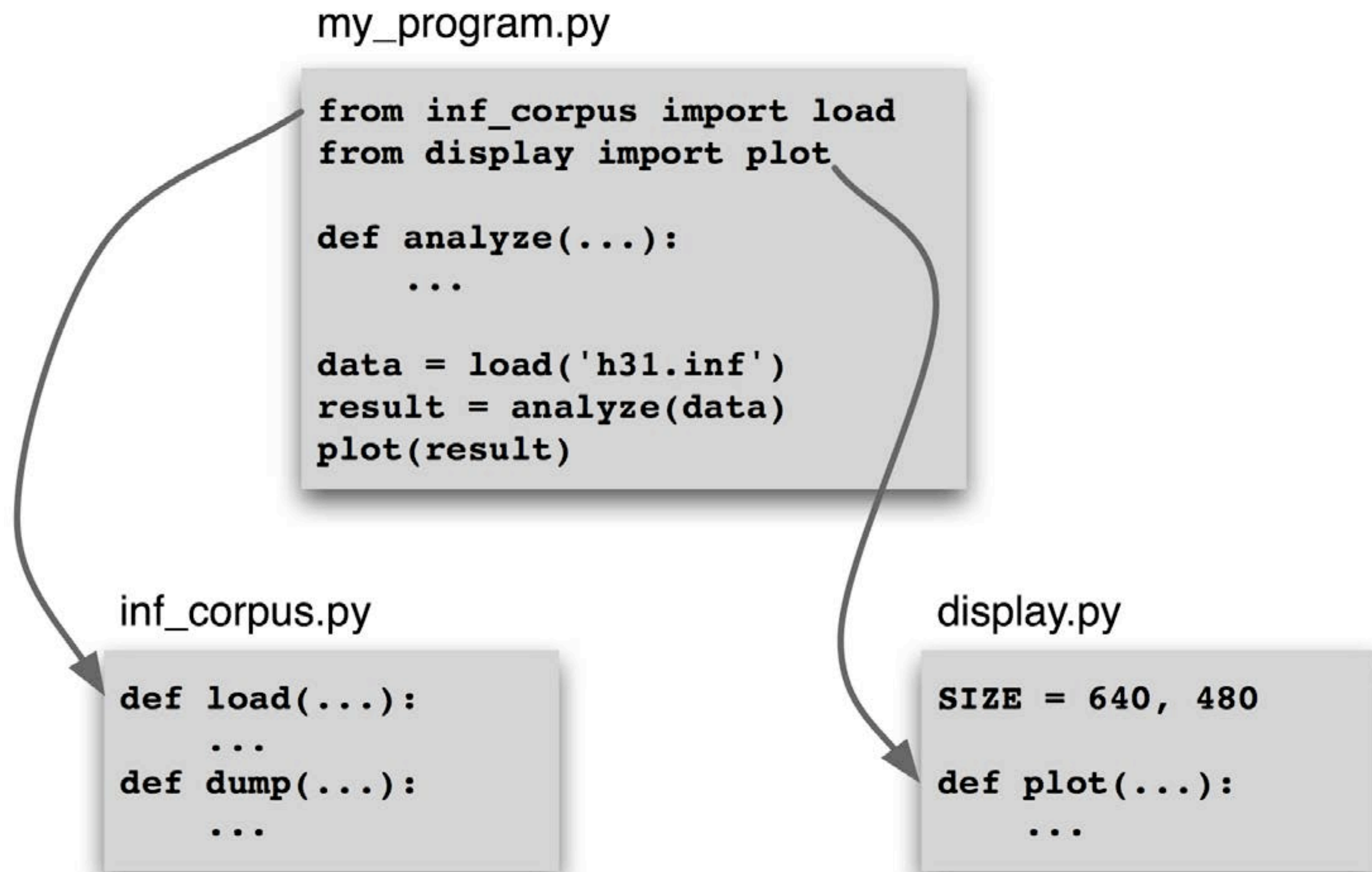
- Module is used to bring logically-related definitions and functions together:
  - The goal is to facilitate re-use and abstraction.
  - An individual `.py` file is a Python module.
    - For example, you can group all your I/O methods.
- The usual structure for a module:
    - Commented lines, e.g. for copyright notice, license information, revision history, etc.
    - Module level docstring
    - Import statements required for the module
    - Global variables
    - A series of function definitions that make up most of the module

Structure of a Python Module

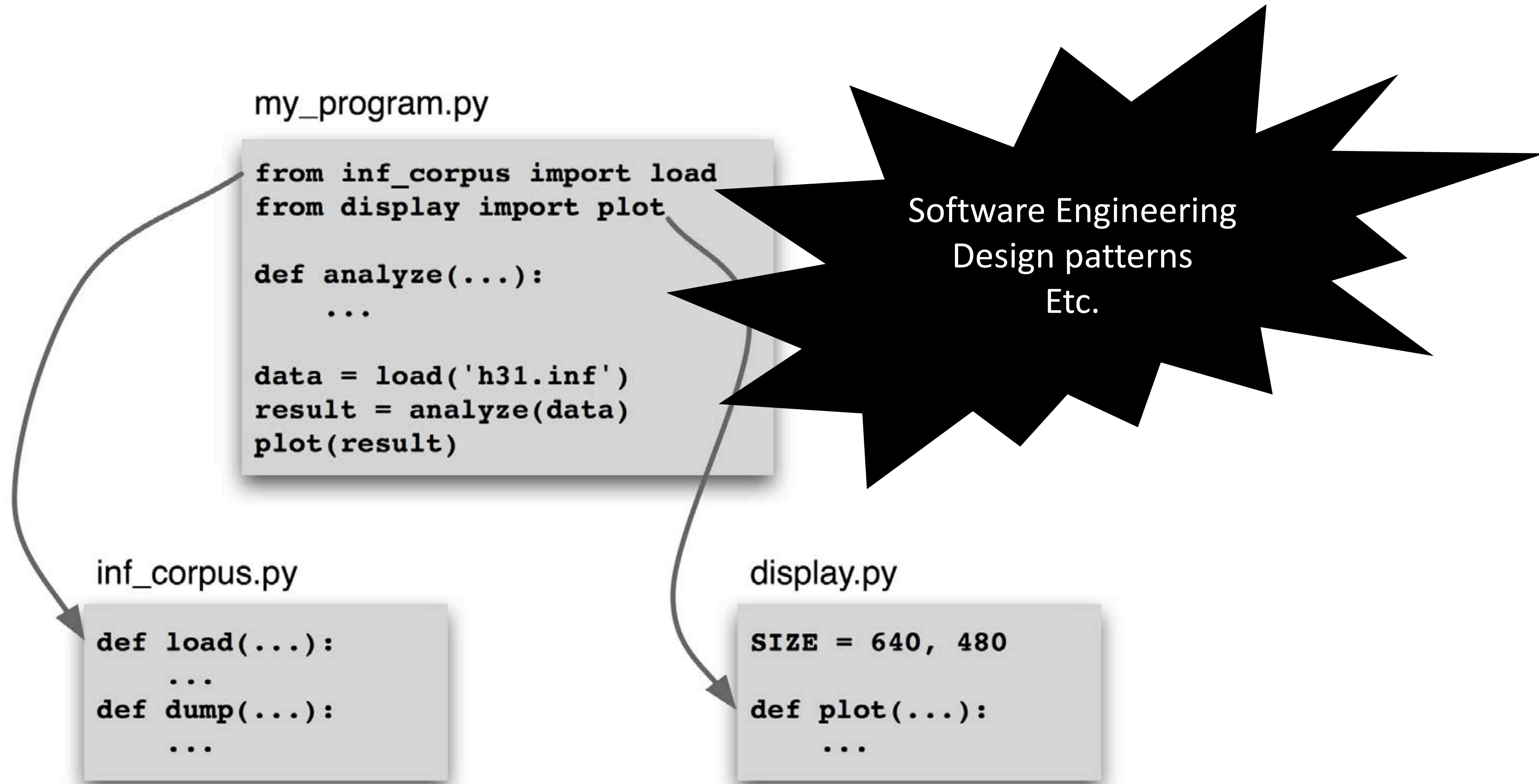
- Module is used to bring logically-related definitions and functions together:
  - The g
  - An in
    - Fo
- The usu
    - Co
    - Mo
    - Import statement   e module
    - Global variables
    - A series of function definitions that make up most of the module

- Some module variables and functions must be only used within the module:
  - Use an underscore in the beginning of their names to hide them, e.g. _helper()
    - These names won't be imported when using `from module import *`
    - List the externally accessible names of a module using a special built-in variable `__all__ = ['method1', 'variablen'].`

Multi-Module Programs



Multi-Module Programs



Packages

- Python has a concept of packages:
  - Think of packages as the directories on a file system and modules as files within directories (there are some important details here).
  - Think of a packages of a special kind module.

Packages

- Python has a concept of packages:
  - Think of packages as the directories on a file system and modules as files within directories (there are some important details here).
  - Think of a packages of a special kind module.
  - Packages are organized hierarchically:
    - Packages may themselves contain subpackages, as well as regular modules.
      - You might have a module called `sys` and a package called `email`, which in turn has a subpackage called `email.mime` and a module within that subpackage called `email.mime.text`.

Packages

- Python has a concept of packages:
  - Think of packages as the directories on a file system and modules as files within directories (there are some important details here).
  - Think of a packages of a special kind module.
  - Packages are organized hierarchically:
    - Packages may themselves contain subpackages, as well as regular modules.
      - You might have a module called `sys` and a package called `email`, which in turn has a subpackage called `email.mime` and a module within that subpackage called `email.mime.text`.

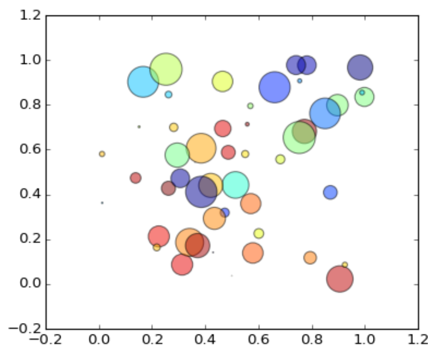
```
import foo # foo imported and bound locally
import foo.bar.baz # foo.bar.baz imported, foo bound locally
import foo.bar.baz as fbb # foo.bar.baz imported and bound as fbb
from foo.bar import baz # foo.bar.baz imported and bound as baz
from foo import attr # foo imported and foo.attr bound as attr
```

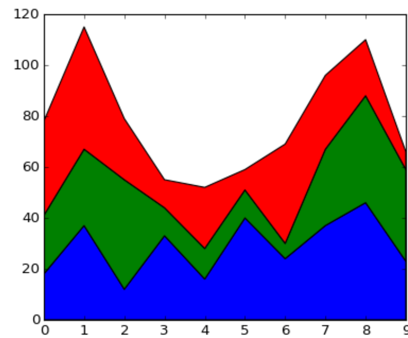
A sample of Python Libraries (Packages)

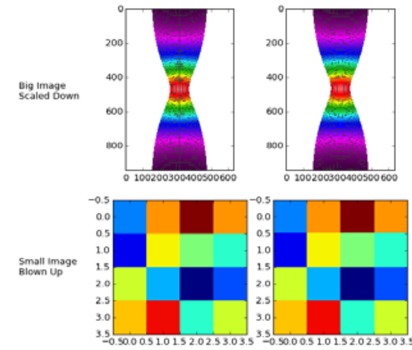
- **Matplotlib**
- **NetworkX**
- **CSV**
- **NumPy**
- …

Matplotlib

- A packge for visualizing data:
  - Sophisticated plotting functions with a MATLAB-style interface;
  - Available from http://matplotlib.sourceforge.net/ .
- You are going to use this library for writing reports and generating results.







Matplotlib

- A packge for visualizing data:
  - Sophisticated plotting functions with a MATLAB-style interface;
  - Available from http://matplotlib.sourceforge.net/ .
- You are going to use this library for writing reports and generating results.
- Browse Matplotlib website at http://matplotlib.org
  - For inspiration, look at the Matplotlib Gallary (/gallery.html)
  - For learning to use the package, look at /examples/api/index.html
  - If you use it, please remember to cite Matplotlib in your final report!

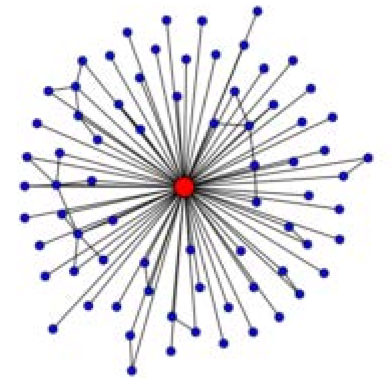
Matplotlib

- A packge for visualizing data:
  - Sophisticated plotting functions with a MATLAB-style interface;
  - Available from http://matplotlib.sourceforge.net/ .
- You are going to use this library for writing reports and generating results.
- Browse Matplotlib website at http://matplotlib.org
  - For inspiration, look at the Matplotlib Gallary (/gallery.html)
  - For learning to use the package, look at /examples/api/index.html
  - If you use it, please remember to cite Matplotlib in your fin

Remember, the goal is to express (then impress).
Use the right plot for presenting your findings.

NetworkX

- The NetworkX package is for defining and manipulating graph.
  - Graph is a structure consists of nodes and edges.
- NetworkX can be used in conjunction with Matplotlib to visualize networks, such as WordNet.
- The NetworkX is avialble from https://networkx.lanl.gov/
- Browse the website for inspiration, code examples, etc.



There are a lot of libraries to use

- **CSV**
  - Python's CSV library can be used to read and write files stored in comma separated values.
- **NumPy**
  - Provides methods for numerical processing in Python.
  - NumPy includes linear algebra functions, which are very useful!
- **PyML**
  - Machine learning in Python.
- **Also look at bindings** for OpenNLP, Gate, Stanford NLP tools, Mallet, …
- **And, even web frameworks (see https://wiki.python.org/moin/WebFrameworks)**
- **Have a look at Python package index** http://pypi.python.org/ before writing something from scratch!

Other important resources

- Always HELP files
- Mailing lists
  - For instance, for NLTK look at https://groups.google.com/forum/#!forum/nltk-users
  - For any other third party libraries that you are going to use there are often a mailing list