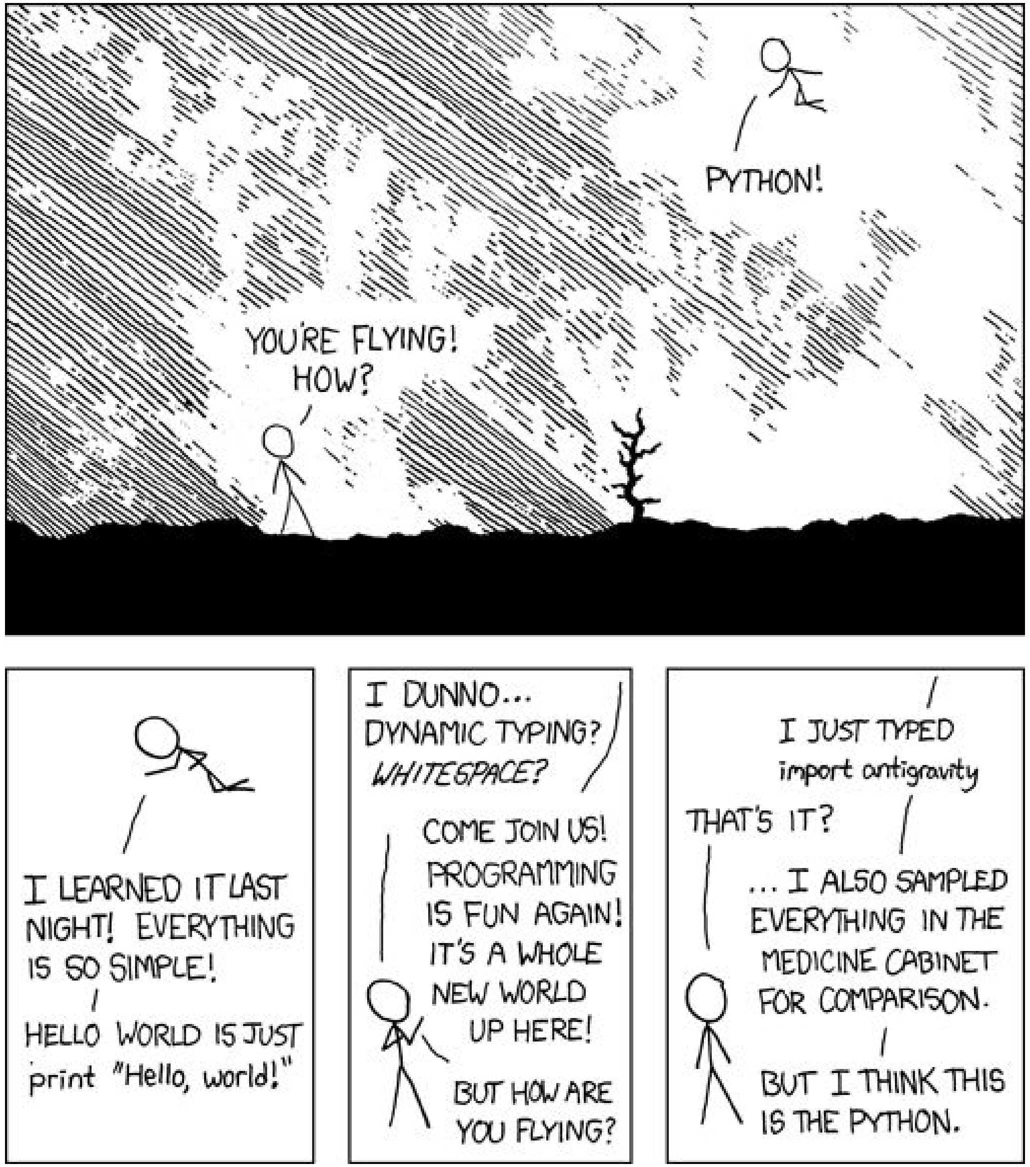
http://xkcd.com/353/



For Loop

A for loop allows us to **go over the items of a sequence such as a list or a string one by one.**

for variable in sequence:

    indented statement(s)

Other statements…

The indented statements (also called the loop body) are executed several times, **once for each item in the sequence.**

For Loop

A for loop allows us to **go over the items of a sequence such as a list or a string one by one.**

for variable in sequence:

    indented statement(s)

Other statements…

Note the colon at the end of the for statement.

For Loop

A for loop allows us to **go over the items of a sequence such as a list or a string one by one.**

for variable in sequence:

    indented statement(s)

Other statements…

The indentation is what determines what statements are executed repeatedly.

For Loop - Iterating over a List

```
cs156 = ['Alice', 'Bob', 'Carol', 'Evan']

for student in cs156:
    print 'Hello', student
print 'Welcome to CS 156'
```

```
for variable in sequence:
    indented statement(s)
Other statements…
```

student is just a variable name. It refers to all the items in the list, **one after the other**.

For Loop - Iterating over a List

```
cs156 = ['Alice', 'Bob', 'Carol', 'Evan']

for student in cs156:
    print 'Hello', student
print 'Welcome to CS 156'
```

for variable in sequence:

    indented statement(s)

Other statements…

Hello Alice

Hello Bob

Hello Carol

Hello Evan

Welcome to CS 156

For Loop - Iterating over a List

```
order = ['apples', 'bananas', 'oranges']

for fruit in order:
    print fruit
```

for variable in sequence:

    indented statement(s)

Other statements…

apples

bananas

oranges

While Loop

```
while condition:
        indented statement(s)
```

While Loop

```
key = 'python'
success  = False        # initialize a boolean variable
while not success:   # use it in the while condition
    password = raw_input('Please enter your password: ')
    if password == key:
        success = True  # update the boolean to exit the loop
print "You're in!"
```

Functions

```
def area(length, width):
    """
    Compute the area of a rectangle
    Parameters:
    length, width (float)
    Returns:
    area (float)
    """
    result = length * width
    return result
```

A function definition starts with the reserved word **def** (short for define), followed by the **function name**.

Functions

```
def area(length, width):
    """
    Compute the area of a rectangle
    Parameters:
    length, width (float)
    Returns:
    area (float)
    """
    result = length * width
    return result
```

Function names are usually lowercase. The function name here is **area**.

Functions

```
def area(length, width):
    """
    Compute the area of a rectangle
    Parameters:
    length, width (float)
    Returns:
    area (float)
    """
    result = length * width
    return result
```

Our function takes two input **parameters:** *length* and *width*.

We don't associate a type with the parameters.

Functions

```
def area(length, width):
    """
    Compute the area of a rectangle
    Parameters:
    length, width (float)
    Returns:
    area (float)
    """
    result = length * width
    return result
```

The function definition header ends with a **colon.**

Functions

```
def area(length, width):
    """
    Compute the area of a rectangle
    Parameters:
    length, width (float)
    Returns:
    area (float)
    """
    result = length * width
    return result
```

Everything in a function definition is indented. There are no curly braces and no begins and ends to delimit the function otherwise.

Functions

```
def area(length, width):
    """
    Compute the area of a rectangle
    Parameters:
    length, width (float)
    Returns:
    area (float)
    """
    result = length * width
    return result
```

A function has its own docstring. It is the first thing in the function. It is also indented.

Functions

```
def area(length, width):
    """
    Compute the area of a rectangle
    Parameters:
    length, width (float)
    Returns:
    area (float)
    """
    result = length * width
    return result
```

result is a variable that is **local** to the function.

Functions

```
def area(length, width):
    """
    Compute the area of a rectangle
    Parameters:
    length, width (float)
    Returns:
    area (float)
    """
    result = length * width
    return result
```

Our function returns the value of the variable *result* which is the product of the length and width.

Lambda Functions

Anonymous functions in Python are called **lambda functions**.

Lambda functions offer a concise way to define functions without giving them a name. They are used to create and use functions on the fly.

Lambda functions are usually **passed as arguments to other functions**.

Lambda Functions

The syntax to specify a lambda function is as follows:

lambda parameters: return_value

The above is a Python **expression**. It can appear in an assignment statement or as an argument to another function.

Lambda Functions

lambda x: x ** 2

We can use the lambda above in the assignment statement:

square = lambda x: x ** 2

Now the variable *square* is a function.

```
>>> square(5)
25
```

The above is equivalent to the following:

```
def square(x):
    return x ** 2
```

Built-in Data Structures: Tuples

A tuple is a **sequence of values separated by commas**.

A tuple has an **arbitrary but finite length**.

Tuples

```
>>> position =  5, 7
>>> type(position)
<type 'tuple'>
>>> print position
(5, 7)
```

Tuples

Tuples may or may not be enclosed in parentheses.

```
>>> position =  (5, 7)
```

tuples are **0 indexed**.

```
>>> print position[0]
5
>>> print position[1]
7
```

Tuples

We can get the length of a tuple:

```
>>> measurements =  5, 7, 5
>>> len(measurements)
3
```

Tuples

Like lists, tuple items may be of different types.

Like lists, tuples may be nested.

```
>>> position =  5, 7
>>> action = (position, 'West', 1)
>>> print action
((5, 7), 'West', 1)
```

Tuples

Tuples are **immutable**.

We cannot append or insert elements in a tuple.

We cannot delete elements from a tuple.

We cannot usually change an element of a tuple.

Tuples

We often **use tuples to assign multiple values at once**.

```
>>> position = (5, 7)
>>> x, y = position
>>> print x
5
>>> print y
7
```

Tuples

```
>>> order = ('apples', 2)
>>> fruit, weight = order
>>> print fruit
apples
>>> print weight
2
```

Tuples

```
>>> action = ((5, 7), 'West', 1)
>>> position, direction, cost = action
>>> print position
(5, 7)
>>> print direction
West
>>> print cost
1
```

Tuples or Lists?

Tuples look a lot like lists except that they have parentheses instead of square brackets.

So how do we decide to use one rather than the other?

Tuples or Lists?

- Tuples are **immutable**, lists are **mutable.**
- Tuples are **faster** than lists.

When we are dealing with **constant data**, tuples make our code **faster** and **safer**.

There is no way to accidentally change a tuple.

Tuples or Lists?

A data structure for the days of the week ?

```
days_of_week = ('Sunday', 'Monday', 'Tuesday',
    'Wednesday', 'Thursday', 'Friday', 'Saturday')
```

A data structure for the names of my friends? We'll assume that I am a fairly sociable and outgoing person.

```
friends = ['Alex', 'Bob', 'Carol', 'Dan']
```

Example: A List of Tuples

Consider a list of points - denoted by their coordinates.

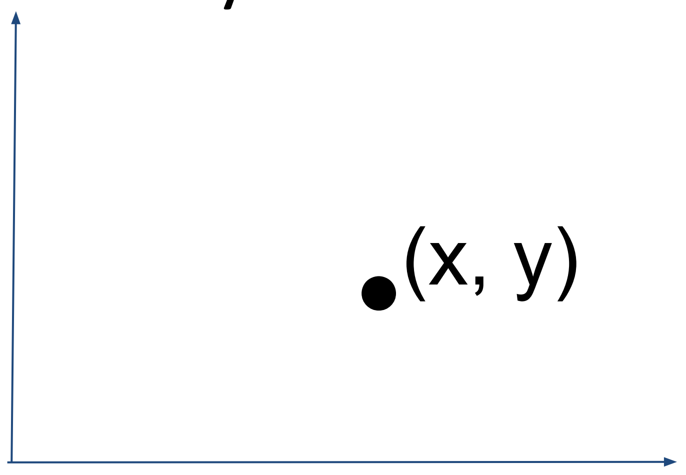
points = [(1, 3), (2, 7), (7, 1), (2, 1)]

Let's write a function, *get_closest_point* that takes such a list of points as a parameter and returns the point that is closest to the origin.

To measure 'closeness' to the origin, we'll use the Manhattan distance between a point and the origin.

Example: A List of Tuples

For a point (x, y) the Manhattan distance to the origin is x + y.



Origin (0, 0)

Example: A List of Tuples

```
def get_closest_point(list_of_points):
    """ return the point that is closest to the origin """
    min_x, min_y = list_of_points[0] # pick the first point
    for x, y in list_of_points[1:]:  # iterate over the rest of the list
        if x + y <= min_x + min_y: # if this point is closer to origin
            min_x, min_y =  x, y
    return min_x, min_y
```

Example: A List of Tuples

```
print get_closest_point([(1, 3), (2, 7), (7, 1), (2, 1)]))
(2, 1)
```

Sorting with a key

The *min*, *max* and *sorted* functions take an optional *key* argument that allows us to specify a function that is used to compute proxy values for each item. The sort is then performed using these proxy values.

The specified *key* function may be a built-in function such as *len*, a user defined function or a lambda function.

Sorting with a key

>>> words = ['in', 'the', 'face', 'of', 'ambiguity', 'refuse', 'the', 'temptation', 'to', 'guess']

To sort the words in the list words from shortest to longest, we can write:

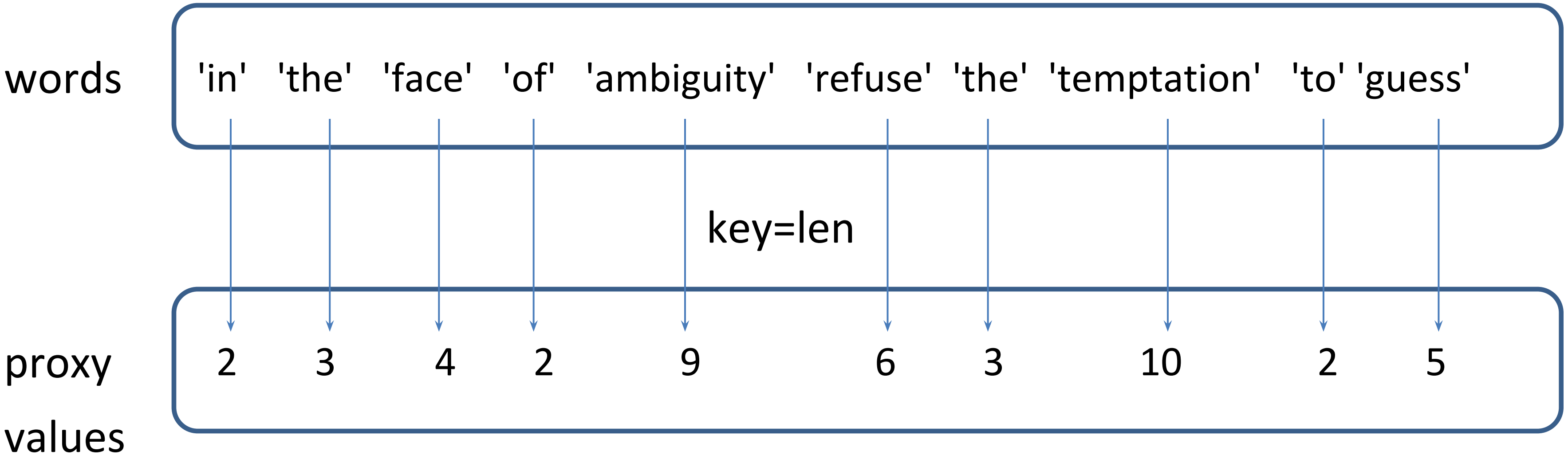
>>> sorted(words, **key=len**)

['in', 'of', 'to', 'the', 'the', 'face', 'guess', 'refuse', 'ambiguity', 'temptation']

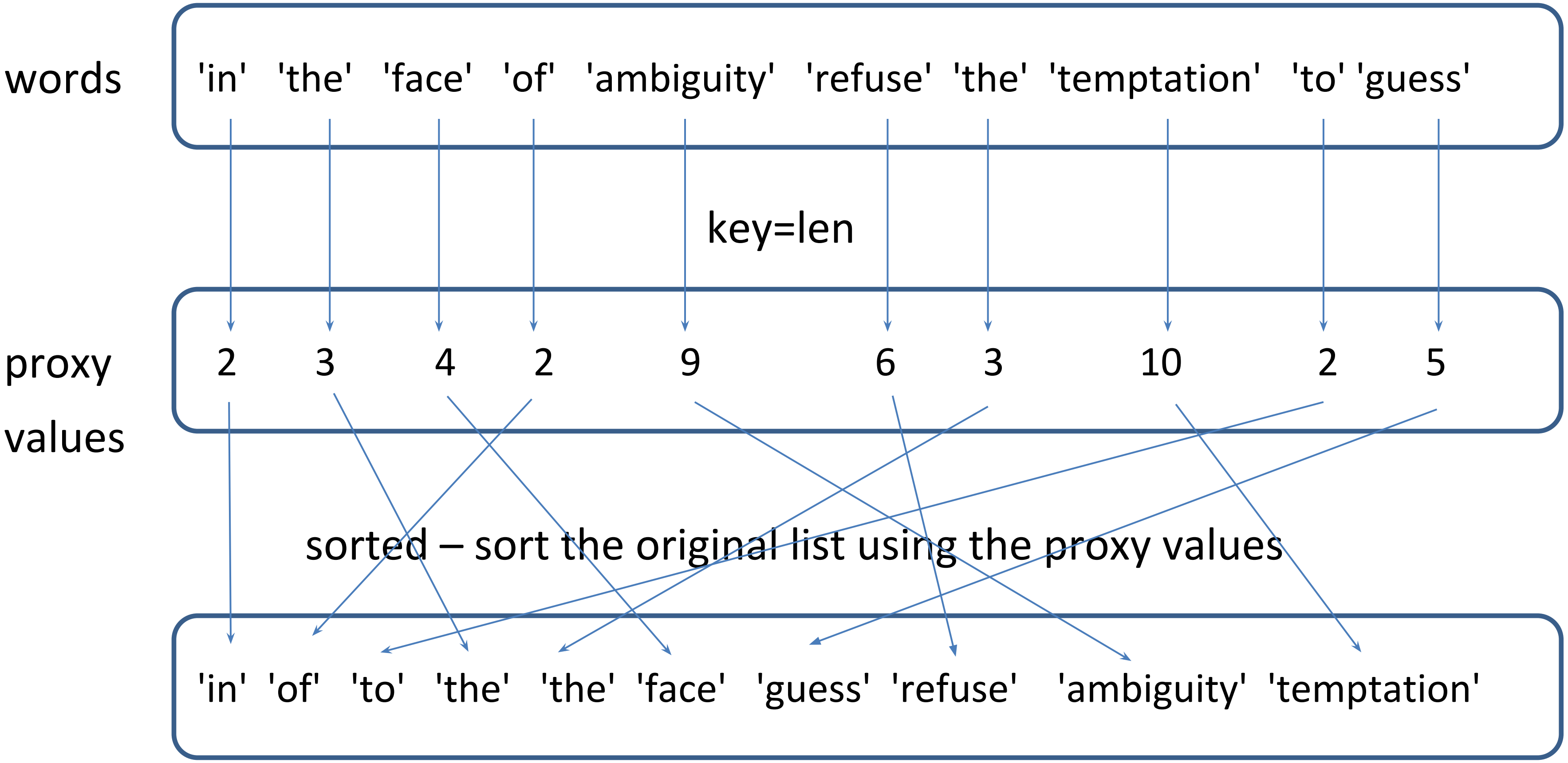
Sorting with a key

words 'in' 'the' 'face' 'of' 'ambiguity' 'refuse' 'the' 'temptation' 'to' 'guess'

Sorting with a key



Sorting with a key



Example: A List of Tuples

Using *min* and *key* with a lambda function:

```
def get_closest_point(list_of_points):
    """ return the point that is closest to the origin"""
    return min(list_of_points, key=lambda coord: sum(coord) )


print get_closest_point([(1, 3), (2, 7), (7, 1), (2, 1)]))
(2, 1)
```

Example: A List of Tuples

list_of_points

[(1, 3), (2, 7), (7, 1), (2, 1)]

key=lambda coord: sum(coord))

4 9 8 3

min?

Built-in Data Structures: Sets

A set is an **unordered collection** of **unique** items.

The items don't need to have the same type but they need to be hashable.

For the data types we have covered so far, immutable and hashable are equivalent.

Sets

We create a set by listing its elements between **curly braces {}**:

```
>>> fruits = {'apple', 'orange', 'banana', 'pear'}
>>> type(fruits)
<type 'set'>
```

Sets

Items of a set are **unique** – even if we enter the same item more than once.

Duplicates are not added.

```
>>> my_set = {4, 6, 4, 'blue'}
>>> my_set
{4, 6, 'blue'}
```

Empty Set

To create an empty set, we have to use set():

```
>>> empty = set()
>>> type(empty)
<type 'set'>
```

Do NOT use {} to create an empty set.

That will create an empty dictionary instead.

Empty Set

To check whether a set is non empty, we use its boolean interpretation.

```
if my_set:      # True only if the set is non empty
    # do something
```

Sets

We can **add** items to a set using the *add* method:

```
>>> sizes = {14, 6, 4, 8}
>>> sizes.add(2)
>>> sizes
{8, 2, 4, 6, 14}
```

Sets

If we try to add a value that already exists in the set, it will do nothing. It won't raise an error.

```
>>> sizes
{8, 2, 4, 6, 14}
>>> sizes.add(6)
>>> sizes
{8, 2, 4, 6, 14}
```

Sets

We can take items out of a set using the ***discard*** method:

```
>>> sizes
{8, 2, 4, 6, 14}
>>> sizes.discard(2)
>>> sizes
{8, 4, 6, 14}
```

Sets

```
>>> sizes
{8, 4, 6, 14}
>>> sizes.discard(10)
>>> sizes
{8, 4, 6, 14}
```

The *discard* method does nothing if the item does not exist in the set.

Sets

We can also take items out of a set using the ***remove*** method:

```
>>> sizes
{8, 4, 6, 14}
>>> sizes.remove(6)
>>> sizes
{8, 4, 14}
```

Sets

```
>>> sizes
{8, 4, 14}
>>> sizes.remove(16)
Traceback (most recent call last):
KeyError: 16
```

The *remove* method raises a KeyError if the item does not exist in the set.

Sets

We can also use ***pop*** **to remove and return an arbitrary item from the set.**

```
>>> sizes
{8, 4, 14}
>>> sizes.pop()
8
>>> sizes
{4, 14}
```

Sets

We can test for **membership** in a set:

>>> fruits = {'apple', 'orange', 'banana', 'pear'}

>>> 'pineapple' **in** fruits

False

>>> 'pear' **in** fruits

True

Sets

We can get the **length of a set:**

```
>>> fruits = {'apple', 'orange', 'banana', 'pear'}
>>> len(fruits)
4
```

Why Sets?

So when do we use a set instead of a list or a tuple?

- **Membership testing is faster with sets.**
- We usually use sets when the **order is not important** and when we are are dealing with **unique items**.
- We'll use a set (*closed* set) in our search algorithms to keep track of the explored states.

Built-in Data Structures: Dictionaries

A dictionary is an **unordered** collection of

key: value pairs.

Each entry contains an index and a value separated by a colon.

In a dictionary, the indices are called **keys**.

The purpose of a dictionary is to store and retrieve values that are **indexed by descriptive keys.**

Dictionaries

We create a dictionary by specifying the key:value pairs between {}:

```
>>> address_book = {'Dan': '555-5678', 'Alice': '555-1234'}
```

Dictionary keys can be of any **hashable** type (strings, numbers, immutable tuples but not lists.)

Keys are unique within a dictionary.

Dictionaries

We **can access the value corresponding to a certain key with the square brackets:**

```
>>> address_book = {'Dan': '555-5678', 'Alice': '555-1234'}
>>> address_book['Alice']
'555-1234'
```

Python **dictionaries are optimized for retrieving the value when we know the key**, but not the other way around.

Dictionaries

We can check for membership in a dictionary, using ***in***:

```
>>> address_book = {'Dan': '555-5678', 'Alice': '555-1234'}
>>> 'Bob' in address_book
False
>>> 'Alice' in address_book
True
```

Dictionaries

We can **add key value pairs to the dictionary by assigning a value to a new key:**

```
>>> address_book = {'Dan': '555-5678', 'Alice': '555-1234'}
>>> address_book['Jim'] = '555-8899'
>>> address_book
{'Alice': '555-1234', 'Dan': '555-5678', 'Jim': '555-8899'}
```

Dictionaries

A dictionary can have at most one value for each key.

Assigning a value to an existing dictionary key replaces the old value with the new one.

```
>>> address_book
{'Alice': '555-1234', 'Dan': '555-5678', 'Jim': '555-8899'}
>>> address_book['Alice'] = '999-3333'
>>> address_book
{'Alice': '999-3333', 'Dan': '555-5678', 'Jim': '555-8899'}
```

Dictionaries

We can use a **for loop** to iterate over a dictionary.

Iterating over a dictionary is equivalent to iterating over its keys:

```
>>> address_book = {'Dan': '555-5678', 'Alice': '555-1234',
                    'Bob': '555-1000'}
>>> for friend in address_book:
...     print(friend)
```

```
Alice
Dan
Bob
```

Dictionaries

Note that since the items in the dictionary are **not ordered**, the friends' names are printed in an arbitrary order.

Dictionaries

To print the phone number associated with each friend:

```
>>> address_book = {'Dan': '555-5678', 'Alice': '555-1234',
                    'Bob': '555-1000'}
>>> for friend in address_book:
...     print(friend, address_book[friend])
```

```
Alice 555-1234
Dan 555-5678
Bob 555-1000
```

Classes and Objects

Although you don't have to write any class definitions in the programming projects, you will have to create some objects and invoke methods defined on these objects.

We'll cover the basics here.

Classes and Objects

```
class Queue:
    "A container with a FIFO queuing policy."
    def __init__(self):
        self.list = []
```

A Python class starts with the reserved word **class**, followed by the class name.

Classes and Objects

```
class Queue:
    "A container with a FIFO queuing policy."
    def __init__(self):
        self.list = []
```

Just like function definitions and compound statements, the class definition header ends with a **colon**.

Classes and Objects

```
class Queue:
    "A container with a FIFO queuing policy."
    def __init__(self):
        self.list = []
```

Everything in a class definition is **indented**. The first line not indented is outside the class.

Classes and Objects

```
class Queue:
    "A container with a FIFO queuing policy."
    def __init__(self):
        self.list = []
```

To create a Queue object, we call *Queue* as if it were a function.

waiting_list = Queue()

Classes and Objects

waiting_list = Queue()

The above statement creates a new instance of the class *Queue* and assigns this object to the local variable *waiting_list*.

There is no explicit *new* operator like in other languages.

Classes and Objects

```
class Queue:
    "A container with a FIFO queuing policy."
    def __init__(self):
        self.list = []
```

The **method that initializes objects** has a special name, __init__ .

Think of it as the **constructor** for the class – even though the object has already been created.

Classes and Objects

```
class Queue:
    '''A container with a FIFO queuing policy.'''
    def __init__(self):
        self.list = []

    def push(self,item):
        '''Enqueue the given item into the queue'
        self.list.insert(0,item)

    def pop(self):
        '''Dequeue and return the earliest enqueu
        return self.list.pop()
```

All method definitions include the special first parameter **self**. Self refers to the object on which the method is invoked.

Classes and Objects

```
class Queue:
    '''A container with a FIFO queuing policy.'''
    def __init__(self):
        self.list = []

    def push(self,item):
        '''Enqueue the given item into the queue'
        self.list.insert(0,item)

    def pop(self):
        '''Dequeue and return the earliest enqueue
        return self.list.pop()
```

All methods have access to the object via the *self* parameter, and so they can all access and manipulate the object's state via the instance variables.

Classes and Objects

```
waiting_list = Queue()
waiting_list.push('Daniel')
waiting_list.push('Anna')
print waiting_list.pop()
'Daniel'
```

The methods are **invoked on an object (*waiting_list*) and the *self* parameter does not have to be specified.**

Imports

In general, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

We can use functions, classes and variables defined in any Python module by executing an **import statement.**

```
import util
```

The module name is the filename name without the .py extension. So when the code is saved in the file 'util.py', the corresponding module name is 'util'.

Namespace and Modules

A **namespace is a collection of identifiers** that belong to a module, to a function, or to a class.

Generally, a namespace holds 'related' things, like all the math functions, or all the date time related behavior.

Each module has its own namespace, so we can use the same identifier name in multiple modules without causing an identification problem.

Imports

To access identifiers belonging to a given module that we have imported, we need to **prefix the identifier with the module name** using the dot notation modulename.identifiername.

We refer to this name as a **fully qualified name.**

Imports

```
>>>import math
```

After importing the math module, we can access the sin function from that module by writing: math.sin

```
>>> math.sin(0)
0.0
```

from … import …

We can also imports names from a module directly into the importing namespace.

>>> **from** math **import** sin

>>> sin(0)

0.0

The *sin* function has been directly imported into the current namespace and may now be used without being prefixed with math.

from … import *

It is also possible to import all names from a module into the current namespace by using the following import statement:

>>> from math import *

This provides an easy way to import all the items from a module into the current namespace.

This statement **should be used sparingly** as the same identifier names may be used in different modules.

Classes and Objects

Let's assume that our Queue class is included in a separate Python module, util.py.

```
import util
waiting_list = util.Queue()
waiting_list.push('Daniel')
waiting_list.push('Anna')
print waiting_list.pop()
'Daniel'
```

Classes and Objects

Let's assume that our Queue class is included in a separate Python module, util.py.

```
from util import Queue
waiting_list = Queue()
waiting_list.push('Daniel')
waiting_list.push('Anna')
print waiting_list.pop()
'Daniel'
```

Classes and Objects

The random module provides a function randint (a, b) that returns a random integer N such that a <= N <= b.

How do I call this function from my own module if I have the following import statement:

import random

A. randint(1, 25)
B. random.randint(1, 25)
C. random(1, 25)
D. randint.random(1, 25)