# Welcome to GKTCS

## IT Training .Consultancy .
## Software Development. Staffing

# Surendra Panpaliya

**Director, GKTCS Innovations Pvt. Ltd, Pune.**

## 16 + Years of Experience ( MCA, PGDCS, BSc. [Electronics] , CCNA)

- Founder,GKTCS Innovations Pvt. Ltd. Pune [ Nov 2009 – Till date ]
- 500 + Corporate Training for HP, IBM, Cisco,Wipro, Samsung etc.
- **Skills**
  - ❑ **Python, Perl,Jython, Django, Android ,**
  - ❑**Ruby, Rail, Cake PHP, LAMP**
  - ❑ Data Communication & Networking, CCNA
  - ❑UNIX /Linux Shell Scripting, System Programming
  - ❑ CA Siteminder, Autosys, SSO, Service Desk, Service Delivery
- Author of 4 Books
- National Paper Presentation Awards at BARC Mumbai

# Agenda

| Day | Module | Topics |
|---|---|---|
| Day 4 | Module 1 | The Python debugger |
| | Module 2 | Decorator, Metaclasses, Generators |
| | Module 3 | Regular Expression |
| | Module 4 | File and Directory Handling, Pickle |

# Module 1
# The Python debugger


# pdb – Interactive Debugger

- pdb implements an interactive debugging environment for Python programs.
- It includes features to let you pause your program, look at the values of variables, and watch program execution step-by-step, so you can understand what your program actually does and find bugs in the logic.

- **From the Command Line**
- The most straightforward way to use the debugger is to run it from the command line, giving it your own program as input so it knows what to run.


# pdb – Interactive Debugger

- **# Pdb_script.py**

```
class MyObj(object):
    def __init__(self, num_loops):
        self.count = num_loops
    def go(self):
        for i in range(self.count):
            print i
        return
if __name__ == '__main__':
    MyObj(5).go()
```

**$ python -m pdb pdb_script.py**

**> .../pdb_script.py(7)<module>()**

**-> """**

**(Pdb)**


# Within the Interpreter

- $ python
- Python 2.7 (r27:82508, Jul 3 2010, 21:12:11) [GCC 4.0.1 (Apple Inc. build 5493)] on darwin Type "help", "copyright", "credits" or "license" for more information.
- >>> import pdb_script
- >>> import pdb
- >>> pdb.run('pdb_script.MyObj(5).go()')
- > <string>(1)<module>()
- (Pdb)


# From Within Your Program

```
# pdb_set_trace.py

import pdb
class MyObj(object):
    def __init__(self, num_loops):
        self.count = num_loops
    def go(self):
        for i in range(self.count):
            pdb.set_trace()
            print i
        return
if __name__ == '__main__':
    MyObj(5).go()
```


# After a Failure

- Debugging a failure after a program terminates is called *post-mortem* debugging. pdb supports post-mortem debugging through the pm() and post_mortem() functions.


# Pdb_post_mortem.py

```
#pdb_post_mortem.py

class MyObj(object):
    def __init__(self, num_loops):
        self.count = num_loops

    def go(self):
        for i in range(self.num_loops):
            print i
        return
```


# Pdb_post_mortem.py

```
#pdb_post_mortem.py

class MyObj(object):
    def __init__(self, num_loops):
        self.count = num_loops

    def go(self):
        for i in range(self.num_loops):
            print i
        return
```


# Pdb_post_mortem.py

- $ python Python 2.7 (r27:82508, Jul 3 2010, 21:12:11) [GCC 4.0.1 (Apple Inc. build 5493)] on darwin Type "help", "copyright", "credits" or "license" for more information.
- >>> from pdb_post_mortem import MyObj
- >>> MyObj(5).go() Traceback (most recent call last): File "<stdin>", line 1, in <module> File "pdb_post_mortem.py", line 13, in go for i in range(self.num_loops): AttributeError: 'MyObj' object has no attribute 'num_loops‘
- >>> import pdb
- >>> pdb.pm()
- > .../pdb_post_mortem.py(13)go()
- -> for i in range(self.num_loops):
- (Pdb)


# Controlling the Debugger

- You interact with the debugger using a small command language that lets you move around the call stack, examine and change the values of variables, and control how the debugger executes your program.
- The interactive debugger uses readline to accept commands.
- Entering a blank line re-runs the previous command again, unless it was a **list** operation.


# Controlling the Debugger

- At any point while the debugger is running you can use **where** (abbreviated **w**) to find out exactly what line is being executed and where on the call stack you are.
- In this case, the module pdb_set_trace.py line 17 in the go() method.
- $ python pdb_set_trace.py
- > .../pdb_set_trace.py(17)go()
- -> print i (Pdb) where
- .../pdb_set_trace.py(21)<module>()
- -> MyObj(5).go()
- > .../pdb_set_trace.py(17)go()
- -> print i


# Controlling the Debugger

- To add more context around the current location, use **list** (**l**).
- (Pdb) list
- 12 self.count = num_loops
- 13
- 14 def go(self):
- 15 for i in range(self.count):
- 16 pdb.set_trace()
- 17 -> print I
- 18 return
- 19
- 20 if __name__ == '__main__':
- 21 MyObj(5).go() [EOF] (Pdb)

# Decorator

# Using decorators for better Python programming


# The pre@mble

- **An intro to reading & writing decorators**
  - Function and method decorators
  - Class decorators
- **Python 2.6 & 2.7 only**
  - No coverage of decorators in Python 3
- Assumes you have **a basic understanding** of:
  - Python!
  - How to write a function
  - How to write a class


# This talk and source code

- Download from:
  - My BitBucket account
  - https://bitbucket.org/gjcross/talks


# *Decorator overview*
# What is a decorator?

- **A function or class that modifies or extends another function or method**
- With some nice syntax
- *That's it!*

- Nothing fancy, nothing new
- Really just syntactic sugar
- Aspect-oriented programming


# *Decorator overview*
# Decorator syntax example

- An example use of a function decorator
- No need to have complex caching code in every function – just decorate them!

```
@cache
def factorise(n):
        factors = []
        # calculate factors of n
        # takes lots of time for large n
        return factors
```


# *Decorator overview*
# Why use decorators?

- Robust design
  - Separation of concerns
  - Can easily turn behaviour on/off
- Improved readability
  - Decorated functions have less baggage
  - Less lines of boilerplate code
  - Less code duplication
  - Simplifies code maintenance
- Widely used in Python libraries & frameworks


# *Decorator overview*
# Why wouldn't you use decorators?

- Not built into Python 2.3 or earlier
  - Important for people maintaining legacy systems
- Can slow your code down
  - Functions nested inside other functions
- Can hamper debugging
- Some tools don't play well with decorators
- Functionality is split between your function and the decorator


# *Decorator overview*
# Decorator syntax

- For functions:
  - Decorators use the syntax @decorator_name
  - @ indicates that it is a decorator
  - Can have arguments (more on that later)
  - Decorators prefix the function with a single line above the function
  - Same for decorating methods and classes


# *Decorator overview*
## In a world without decorators

```
def acquire_image(x0,y0,resolution):
    check_valid_origin(x0, y0)
    check_valid_resolution(resolution)lock_camera()
    log_event(x0, y0, resolution)
    camera.setup(x0, y0, resolution)
    image = camera.read()
    unlock_camera()
    assert_valid_image(image)
    return image
```


# *Decorator overview*
## In a world with decorators

```
@assert_inputs
@log_event
@validate_image
def acquire_image(x0,y0,resolution):
    with camera.lock:
        camera.setup(x0, y0, resolution)
        image = camera.read()
    return image
```


# *Decorator overview*
# Typical uses…

- Pre-conditions and post-conditions
  - Assert types and/or values
  - Check returned values
- Cache results
  - Network data
  - Expensive computations


# *Decorator overview*
# …more typical uses

- Debugging, logging, tracing and profiling:
  - Record function results
  - Entry and exit times
  - States
  - Returned values
- Simplify synchronisation and/or locking of resources
  - Databases, threads, serial ports, hardware


# *Decorator overview*
# The standard Python decorators

- Commonly used standard decorators:
  - @property
  - @classmethod
  - @staticmethod


# *Decorator overview*
# The standard Python decorators

- staticmethod(function) → function

- The @staticmethod decorator modifies a method function so that it does not use any self variable. The method function will not have access to a specific instance of the class.

- This kind of method is part of a class, but can only be used when qualified by the class name or an instance variable.


# *Decorator overview*
# The standard Python decorators

- classmethod(function) → function
- The @classmethod decorator modifies a method function so that it receives the class object as the first parameter instead of an instance of the class. This method function will have access to the class object itself.

- property(fget[, fset, fdel, doc]) → function
- The @property decorator modifies from one to three method functions to be a properties of the class. The returned method functions invokes the given getter, setter and/or deleter functions when the attribute is referenced.
- Demo : decorator1.py


# Defining Decorators

- A decorator is a function which accepts a function and returns a new function. Since it's a function, we must provide three pieces of information: the name of the decorator, a parameter, and a suite of statements that creates and returns the resulting function.

- The suite of statements in a decorator will generally include a function def statement to create the new function and a return statement.

- A common alternative is to include a class definition statement . If a class definition is used, that class must define a callable object by including a definition for the __call__() method and (usually) being a subclass of collections.Callable.


# Defining Decorators

- There are two kinds of decorators, decorators without arguments and decorators with arguments. In the first case, the operation of the decorator is very simple. In the case where the decorator accepts areguments the definition of the decorator is rather obscure.
- A simple decorator has the following outline:
- def myDecorator( argumentFunction ):
- def resultFunction( *args, **keywords ):
- enhanced processing including a call to argumentFunction
- resultFunction.__doc__= argumentFunction.__doc__
- return resultFunction


# Defining Decorators

- In some cases, we may replace the result function definition with a result class definition to create a callable class.

- Here's a simple decorator that we can use for debugging. This will log function entry, exit and exceptions.

- Example : Trace.py


# Defining Decorators

1. The result function, loggedFunc(), is built when the decorator executes. This creates a fresh, new function for each use of the decorator.
2. Within the result function, we evaluate the original function. Note that we simply pass the argument values from the evaluation of the result function to the original function.
3. We move the original function's docstring and name to the result function. This assures us that the result function looks like the original function.


# Defining Decorators

Here's a class which uses our @trace decorator.

trace_client.py

```
class MyClass( object ):
    @trace
    def __init__( self, someValue ):
        """Create a MyClass instance."""
        self.value= someValue
    @trace
    def doSomething( self, anotherValue ):
        """Update a value."""
        self.value += anotherValue
```


# Defining Decorators

Here's a class which uses our @trace decorator.

trace_client.py

```
class MyClass( object ):
    @trace
    def __init__( self, someValue ):
        """Create a MyClass instance."""
        self.value= someValue
    @trace
    def doSomething( self, anotherValue ):
        """Update a value."""
        self.value += anotherValue
```


# Defining Complex Decorators

- A decorator transforms an argument function definition into a result function definition. In addition to a function, we can also provide argument values to a decorator. These more complex decorators involve a two-step dance that creates an intermediate function as well as the final result function.

- The first step evaluates the abstract decorator to create a concrete decorator. The second step applies the concrete decorator to the argument function. This second step is what a simple decorator does.

- Assume we have some qualified decorator, for example @debug( flag ), where flag can be True to enable debugging and False to disable debugging. Assume we provide the following function definition.


# Defining Complex Decorators

```
debugOption= True
class MyClass( object ):
    @debug( debugOption )
    def someMethod( self, args ):
        real work
```

1. Here's what happens when Python creates the definition of the someMethod() function.
2. Defines the argument function, someMethod().
3. Evaluate the abstract decorator debug( debugOption ) to create a concrete decorator based on the argument value.
4. Apply the concrete decorator the the argument function, someMethod().
5. The result of the concrete decorator is the result function, which is given the name someMethod().


# Defining Complex Decorators

- Here's an example of one of these more complex decorators.
- #debug1.py
- def debug( theSetting ):
- def concreteDescriptor( aFunc ):
- if theSetting:
- def debugFunc( *args, **kw ):
- print "enter", aFunc.__name__
- return aFunc( *args, **kw )
- debugFunc.__name__= aFunc.__name__
- debugFunc.__doc__= aFunc.__doc__
- return debugFunc
- else:
- return aFunc
- return concreteDescriptor


# Decorator Exercises

1. Merge the @trace and @debug decorators. Combine the features of the @trace decorator with the parameterization of the @debug decorator. This should create a better @trace decorator which can be enabled or disabled simply.
2. Create a @timing decorator. Similar to the parameterized @debug decorator, the @timing decorator can be turned on or off with a single parameter. This decorator prints a small timing summary


# *Decorator overview*
# Some other framework examples

- unittest
  - @skipUnless
- Django
  - @login_required
- Bottle
  - @route
- Fabric, automated build & deployment
  - @roles
  - @runs_once


# Class decorators

- Added in Python 2.6 and 3.0
- PEP 3129
- Simpler than metaclasses
- Class decorators are not inherited
  - Easier to control
  - Metaclasses are inherited


# *Class decorators*
# Examples

- Add or override methods to classes
- Singletons
- Class checks:
    - Must have unit tests
    - Must have specific methods implemented
- Register classes; eg.
    - Implements a specific interface
    - Controls hardware, needs additional safety testing


# Some advice

- No hidden surprises
  - Do one thing and do it well
  - A clear name
  - No side effects
- Make sure your decorator stacks nicely
- Don’t overuse decorators!
- Don’t confuse Python decorators with the classic Decorator design pattern


# Metaclasses

- A **metaclass** is a class whose instances are classes.
- Just as an ordinary class defines the behavior of certain objects, a metaclass defines the behavior of certain classes and their instances.
- Each language has its own metaobject protocol, a set of rules that govern how objects, classes, and metaclasses interact
-


# Metaclasses

- **Classes as objects**
- Before understanding metaclasses, you need to master classes in Python. And Python has a very peculiar idea of what classes are, borrowed from the Smalltalk language.
- In most languages, classes are just pieces of code that describe how to produce an object. That's kinda true in Python too:


# Metaclasses

- >>> class ObjectCreator(object):
- ... pass ...
- >>> my_object = ObjectCreator()
- >>> print(my_object)
- <__main__.ObjectCreator object at 0x8974f2c>


# Metaclasses

- In Python, the builtin class type is a metaclass. Consider this simple Python class:


# Metaclasses

```
class Car(object):
    __slots__ = ['make', 'model', 'year', 'color']

    def __init__(self, make, model, year, color):
        self.make = make
        self.model = model
        self.year = year
        self.color = color

    @property
    def description(self):
        """ Return a description of this car. """
        return "%s %s %s %s" % (self.color, self.year, self.make, self.model)
```


# Metaclasses

- At run time, Car itself is an instance of type.
- The source code of the Car class, shown above, does not include such details as the size in bytes of Car objects, their binary layout in memory, how they are allocated, that the __init__ method is automatically called each time a Car is created, and so on.


# Metaclasses

- These details come into play not only when a new Car object is created, but also each time any attribute of a Car is accessed.
- In languages without metaclasses, these details are defined by the language specification and can't be overridden. In Python, the metaclass, type, controls these details of Car's behavior. They can be overridden by using a different metaclass instead of type.


# Metaclasses

- The above example contains some redundant code to do with the four attributes make, model, year, and color.
- It is possible to eliminate some of this redundancy using a metaclass.
- In Python, a metaclass is most easily defined as a subclass of type.


# Metaclasses

```
class AttributeInitType(type):
    def __call__(self, *args, **kwargs):
        """ Create a new instance. """
        # First, create the object in the normal default way.
        obj = type.__call__(self, *args)

        # Additionally, set attributes on the new object.
        for name, value in kwargs.items():
            setattr(obj, name, value)

        # Return the new object.
        return obj
```


# Metaclasses

- This metaclass only overrides object creation. All other aspects of class and object behavior are still handled by type.
- Now the class Car can be rewritten to use this metaclass. This is done in Python 2 by assigning to __metaclass__ within the class definition (in Python 3 you provide a named argument, metaclass=M to the class definition instead):


# Metaclasses

```
class Car(object):
    __metaclass__ = AttributeInitType
    __slots__ = ['make', 'model', 'year', 'color']

    @property
    def description(self):
        """ Return a description of this car. """
        return "%s %s %s %s" % (self.color, self.year, self.make,
     self.model)
```


# Metaclasses

- Car objects can then be instantiated like this:
- cars = [ Car(make='Toyota', model='Prius', year=2005, color='green'),

  Car(make='Ford', model='Prefect', year=1979, color='blue')]


# Metaclasses

- Everything is an Object
- Everything is an object
- Everything has a type
- No real difference between 'class' and 'type'
- Classes are objects
- Their type is type


# Metaclasses

- Honestly, it's True
- >>> **class** Something(object):
  ... **pass**
  ...
  >>> Something
  <**class** '__main__.Something'>
  >>> type(Something)
  <type 'type'>


# Metaclasses

- Here we can see that a class created at the interactive interpreter is a first class object.
- The Class of a Class is...
- Its metaclass...
- Just as an object is an instance of its class; a class is an instance of its metaclass.
- The metaclass is called to create the class.
- *In exactly the same way as any other object in Python.*


# Metaclasses

- So when you create a class...
- The interpreter calls the metaclass to create it...
- For a normal class that inherits from object this means that type is called to create the class:
- >>> help(type) Help on class type in module __builtin__: class type(object) | type(object) -> the object's type | type(name, bases, dict) -> a new type


# Metaclasses

- It is this second usage of type that is important. When the Python interpreter executes a class statement (like in the example with the interactive interpreter from a couple of sections back), it calls type with the following arguments:
- The name of the class as a string
- A tuple of base classes - for our example this is the 'one-pl' [1] (object,)
- A dictionary containing members of the class (class attributes, methods, etc) mapped by their names


# Metaclasses

```
>>> def __init__(self):
...     self.message = 'Hello World'
...
>>> def say_hello(self):
...     print self.message
...
>>> attrs = {'__init__': __init__, 'say_hello': say_hello}
>>> bases = (object,)
>>> Hello = type('Hello', bases, attrs)
>>> Hello
<class '__main__.Hello'>
>>> h = Hello()
>>> h.say_hello()
Hello World
```


# Metaclasses

The Magic of __metaclass__

- We can provide a custom metaclass by setting __metaclass__ in a class definition to *any callable that takes the same arguments as type*.
- The normal way to do this is to inherit from type:

- ```
  class PointlessMetaclass(type):
      def __new__(meta, name, bases, attrs):
          # do stuff...
          return type.__new__(meta, name, bases, attrs)
  ```
  ```


# Metaclasses

- In Action...
- >>> **class** WhizzBang(object):
  ... __metaclass__ = PointlessMetaclass
  ...
  >>> WhizzBang
  <**class** '__main__.WhizzBang'>
  >>> type(WhizzBang)
  <**class** '__main__.PointlessMetaClass'>


# Metaclasses

- What can we do with this?
- Well (I'm glad you asked)... our metaclass will be called whenever a new class is created that uses it. Here are some ideas:
- Decorate all methods in a class for logging, or profiling.
- Automatically mix-in new methods.
- Register classes as they are created. (Auto-register plugins or create a db schema from class members for example.)
- Provide interface registration, auto-discovery of features and interface adaptation.
- Class verification: prevent subclassing, verify all methods have docstrings.

# Regular expressions

Simple patterns
Using regular expression
Pattern power
Modifying string


# Regular expressions

- Regular expressions are a powerful and standardized way of searching, replacing, and parsing text with complex patterns of characters.

- REs are strings containing text and special characters that describe a pattern with which to recognize multiple strings.


# Introduction

- \b[A-Z0-9._%-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b
- What is this?
- Is it a language?
- Is it an engine?


# Introduction (cont…)

- To get complete information from incomplete data in hand
- Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C


# Special Symbols and Characters

| Notation | Description | Example RE |
|---|---|---|
| Symbols | | |
| literal | Match literal string value literal | foo |
| re1\|re2 | Match regular expressions re1 or re2 | foo\|bar |
| . | Match any character (except NEWLINE) | b.b |
| ^ | Match start of string | ^Dear |
| $ | Match end of string | /bin/*sh$ |
| * | Match 0 or more occurrences of preceding RE | [A-Za-z0-9]* |
| + | Match 1 or more occurrences of preceding RE | [a-z]+\.com |
| ? | Match 0 or 1 occurrence(s) of preceding RE | goo? |
| {N} | Match N occurrences of preceding RE | [0-9]{3} |


# Special Symbols and Characters

| | | |
|---|---|---|
| {M,N} | Match from M to N occurrences of preceding RE | [0-9]{5,9} |
| [...] | Match any single character from character class | [aeiou] |
| [..x-y..] | Match any single character in the range from x to y | [0-9],[A-Za-z] |
| [^...] | Do not match any character from character class, including any ranges, if present | [^aeiou], [^A-Za-z0-9_] |
| (*\|+\|?\|{})? | Apply "non-greedy" versions of above occurrence/repetition symbols ( *, +, ?, {}) | .*?[a-z] |
| (...) | Match enclosed RE and save as subgroup | ([0-9]{3})?, f(oo\|u)bar |


# Special Characters

| Notation | Description | Example RE |
|---|---|---|
| \d | Match any decimal digit, same as [0-9](\D is inverse of \d: do not match any numeric digit) | data\d+.txt |
| \w | Match any alphanumeric character, same as [A-Za-z0-9_] (\W is inverse of \w) | [A-Za-z_]\w+ |
| \s | Match any whitespace character, same as [ \n\t\r\v\f] (\S is inverse of \s) | of\sthe |
| \b | Match any word boundary (\B is inverse of \b) | \bThe\b |
| \nn | Match saved subgroup nn (see (...) above) | price: \16 |
| \c | Match any special character c verbatim (i.e., with out its special meaning, literal) | \., \\, * |
| \A (\Z) | Match start (end) of string (also see ^ and $ above) | \ADear |


# Matching More Than One RE Pattern with Alternation ( | )

| RE Pattern | Strings Matched |
|---|---|
| at\|home | at, home |
| r2d2\|c3po | r2d2, c3po |
| bat\|bet\|bit | bat, bet, bit |


# Matching Any Single Character ( .)

| RE Pattern | Strings Matched |
|---|---|
| f.o | Any character between "f" and "o", e.g., fao, f9o, f#o, etc. |
| .. | Any pair of characters |
| .end | Any character before the string end |

Q: What if I want to match the dot or period character?
A: In order to specify a dot character explicitly, you must escape its functionality with a backslash, as in "\.".

# Matching from the Beginning or End of Strings or Word Boundaries ( ^/$ /\b /\B)

- To match a pattern starting from the beginning, you must use the carat symbol ( ^ ) or the special character \A (backslash-capital "A").
- The latter is primarily for keyboards that do not have the carat symbol, i.e., international. Similarly, the dollar sign ( $ ) or \Z will match a pattern from the end of a string.

| **RE Pattern** | **Strings Matched** |
|---|---|
| ^From | Any string that starts with From |
| /bin/tcsh$ | Any string that ends with /bin/tcsh |
| ^Subject: hi$ | Any string consisting solely of the string Subject: hi |


# Creating Character Classes ( [ ] )

| RE Pattern | Strings Matched |
|---|---|
| b[aeiu]t | bat, bet, bit, but |
| [cr][23][dp][o2] | A string of 4 characters: first is "r" or "c," then "2" or "3," followed by "d" or "p," and finally, either "o" or "2," e.g., c2do, r3p2, r2d2, c3po, etc. |


# Denoting Ranges ( - ) and Negation ( ^ )

| **RE Pattern** | **Strings Matched** |
|---|---|
| z.[0-9] | "z" followed by any character then followed by a single digit |
| [r-u][env-y] | "r" "s," "t" or "u" followed by "e," "n," "v," "w," "x," or "y" |
| [us] | followed by "u" or "s" |
| [^aeiou] | A non-vowel character (Exercise: Why do we say "non-vowels" rather than "consonants"?) |
| [^\t\n] | Not a TAB or NEWLINE |
| ["-a] | In an ASCII system, all characters that fall between "" and "a," i.e., between ordinals 34 and 97 |


# Multiple Occurrence/Repetition Using Closure Operators ( *, +, ?, { } )

| RE Pattern | Strings Matched |
|---|---|
| [dn]ot? | "d" or "n," followed by an "o" and, at most, one "t" after that, i.e., do, no, dot, not |
| 0?[1-9] | Any numeric digit, possibly prepended with a "0," e.g., the set of numeric representations of the months January to September, whether single- or double-digits |
| [0-9]{15,16} | Fifteen or sixteen digits, e.g., credit card numbers |
| </?[^>]+> | Strings that match all valid (and invalid) HTML tags |
| [KQRBNP][a-h][1-8]-[a-h][1-8] | Legal chess move in "long algebraic" notation (move only, no capture, check, etc.), i.e., strings which start with any of "K," "Q," "R," "B," "N," or "P" followed by a hyphenated-pair of chess board grid locations from "a1" to "h8" (and everything in between), with the first coordinate indicating the former position and the second being the new position. |


# Special Characters Representing Character Sets

| RE Pattern | Strings Matched |
|---|---|
| \w+-\d+ | Alphanumeric string and number separated by a hyphen |
| [A-Za-z]\w* | Alphabetic first character, additional characters (if present) can Be alphanumeric (almost equivalent to the set of valid Python identifiers |
| \d{3}-\d{3}-\d{4} | (American) telephone numbers with an area code prefix, as in 800-555-1212 |
| \w+@\w+\.com | Simple e-mail addresses of the form XXX@YYY.com |


# Designating Groups with Parentheses ( ( ) )

- A pair of parentheses ( ( ) ) can accomplish either (or both) of the below when used with regular expressions:
  - Grouping regular expressions
  - Matching subgroups

| RE Pattern | Strings Matched |
|---|---|
| \d+(\.\d*)? | Strings representing simple floating point number, that is, any number of digits followed optionally by a single decimal point and zero or more numeric digits, as in "0.004," "2," "75.," etc. |
| (Mr?s?\. )?[A-Z][a-z]* [ A-Za-z-]+ | First name and last name, with a restricted first name (must start with uppercase; lowercase only for remaining letters, if any), the full name prepended by an optional title of "Mr.," "Mrs.," "Ms.," or "M.," and a flexible last name, allowing for multiple words, dashes, and uppercase letters |


# REs and Python

- The re module was introduced to Python in version 1.5.
- For older version of Python, you will have to use the now-obsolete regex and regsub modules
- Both modules were removed from Python in 2.5


# Common Regular Expression Functions and Methods

| Function/Method | Description |
|---|---|
| re Module Function Only | |
| compile(pattern, flags=0) | Compile RE pattern with any optional flags and return a regex object |
| re Module Functions and regex Object Methods | |
| match(pattern, string, flags=0) | Attempt to match RE pattern to string with optional flags; return match object on success, None on failure |
| search(pattern, string, flags=0) | Search for first occurrence of RE pattern within string with optional flags; return match object on success, None on failure |


# Common Regular Expression Functions and Methods

| | |
|---|---|
| findall(pattern, string[,flags]) | Look for all (non-overlapping) occurrences of pattern in string; return a list of matches |
| finditer(pattern, string[, flags]) | Same as findall() except returns an iterator instead of a list; for each match, the iterator returns a match object |
| split(pattern, string, max=0) | Split string into a list according to RE pattern delimiter and return list of successful matches, splitting at most max times (split all occurrences is the default) |
| sub(pattern, repl, string, max=0) | Replace all occurrences of the RE pattern in string with repl, substituting all occurrences unless max provided (also see subn() which, in addition, returns the number of substitutions made) |


# Common Regular Expression Functions and Methods

| **Match Object Methods** | |
|---|---|
| group(num=0) | Return entire match (or specific subgroup num) |
| groups() | Return all matching subgroups in a tuple (empty if there weren't any) |


# Compiling Regular Expressions

```
>>> import re
>>> p = re.compile('ab*')
>>> print p
<re.RegexObject instance at 80b4150>
```

re.compile() also accepts an optional *flags* argument

```
>>> p = re.compile('ab*', re.IGNORECASE)
```


# Performing Matches

- match():Determine if the RE matches at the beginning of the string.

- search(): Scan through a string, looking for any location where this RE matches.


# Performing Matches

- findall() :Find all substrings where the RE matches, and returns them as a list.

- finditer() :Find all substrings where the RE matches, and returns them as an iterator.


# What they return

- match() and search() return None if no match can be found.
- If they're successful, a MatchObject instance is returned


# Usage

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
<_sre.SRE_Pattern object at 80c3c28>
>>> p.match("")
>>> print p.match("")
None
>>> m = p.match( 'tempo')
>>> print m
<_sre.SRE_Match object at 80c4f68>
```


# Querying matchobject

- `group()` Return the string matched by the RE
- `start()` Return the starting position of the match
- `end()` Return the ending position of the match
- `span()` Return a tuple containing the (start, end) positions of the match


# Examples

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```


# If match is at the middle

```
>>>print p.match('::: message')
None
>>>m = p.search('::: message')
          ; print m
<re.MatchObject instance at 80c9650>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```


# In script

```
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
    print 'Match found: ', m.group()
else:
    print 'No match'
```


# To find all matches

```
>>> p = re.compile('\d+')
>>> p.findall('12 drummers
  drumming, 11 pipers piping, 10
  lords a-leaping')
['12', '11', '10']
```


# Module-Level Functions

```
>>> print re.match(r'From\s+',
  'Fromage amk')
None
>>> re.match(r'From\s+', 'From
  amk Thu May 14 19:12:10 1998')
<re.MatchObject instance at
  80c5978>
```


# Compilation Flags

| Flag | Meaning |
|---|---|
| DOTALL, S | Make . match any character, including newlines |
| IGNORECASE, I | Do case-insensitive matches |
| LOCALE, L | Do a locale-aware match |
| MULTILINE, M | Multi-line matching, affecting ^ and $ |
| VERBOSE, X | Enable verbose REs, which can be organized more cleanly and understandably. |


# Pattern powers

- |
- $
- ^
- \b
- \B


# Grouping

```
>>> p = re.compile('(ab)*')
>>> print p.match('abababababab').span()
(0, 10)
```


# Grouping

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```


# Non-capturing and Named Groups

```
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```


# (?=...)

Positive lookahead assertion. This succeeds if the contained regular expression, represented here by ..., successfully matches at the current location, and fails otherwise. But, once the contained expression has been tried, the matching engine doesn't advance at all; the rest of the pattern is tried right where the assertion started.


# (?!...)

Negative lookahead assertion. This is the opposite of the positive assertion; it succeeds if the contained expression ***doesn't*** match at the current position in the string.


# What are the following?

- .*[.].*$
- .*[.](?!bat$).*$
- .*[.](?!bat$|exe$).*$


# Modifying strings

| Method/ Attribute | Purpose |
|---|---|
| split() | Split the string into a list, splitting it wherever the RE matches |
| sub() | Find all substrings where the RE matches, and replace them with a different string |
| subn() | Does the same thing as sub(), but returns the new string and the number of replacements |


# Splitting Strings

```
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and
  sweet, of split().')
['This', 'is', 'a', 'test', 'short',
  'and', 'sweet', 'of', 'split', '']
>>> p.split('This is a test, short and
  sweet, of split().', 3)
['This', 'is', 'a', 'test, short and
  sweet, of split().']
```


# Know the delimeter

```
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
>>> p.split('This... is a
  test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a
  test.')
['This', '... ', 'is', ' ', 'a',
  ' ', 'test', '.', '']
```


# Search and Replace

```
>>> p = re.compile( '(blue|white|red)')
>>> p.sub( 'colour', 'blue socks and red
  shoes')
'colour socks and colour shoes'
>>> p.sub( 'colour', 'blue socks and red
  shoes', count=1)
'colour socks and red shoes'
```


# Debugging

- Kodos
- Site:http://kodos.sourceforge.net/


# Case studies

- Street address
- URL
- A valid C variable
- Phone numbers
- Cell numbers

# File and Directory handling


# File and Directory handling

- File I/O operations
- * Built-in file and directory handling libraries
- * fileinput
- * stat
- * filecmp and dircmp
- * glob, zipfile and tarfile
- * pickle and shelve modules
- * Serialization using json


# File and Directory handling

- **Built-in Functions**
- open(*filename*[, *mode*][, *buffering*]) → file object
- Create a Python file object associated with an operating system file.

- file(*filename*[, *mode*][, *buffering*]) → file object
- mode can be ‘r’, ‘w’ , ‘a’, ‘r+’, ‘w+’ etc


# File and Directory handling

- **Examples**. The following examples create file objects for further processing:
- myLogin = open( ".login", "r" )
- newSource = open( "somefile.c", "w" )
- theErrors = open( "error.log", "a" )
- someData = open( 'source.dat', 'rb' )


# File Statement

- File with With statement

```
with file("somefile","r") as source:
        for line in source: print line
```

- At the end of the **with** statement, irrespective of any exceptions which are handled – or not handled – the file will be closed and the relevant resources released.


# File Statement

- file.read([*size*]) → string
- file.readline([*size*]) → string
- file.readlines([*hint*]) → list of strings
- file.flush()
- file.write(*string*)
- file.writelines(*list*)
- file.truncate([*size*])


# File Statement

- file.seek(*offset*[, *whence*])
- file.tell() → integer
- file.close()
- file.fileno() → integer
- file.isatty() → boolean
- file.closed -> boolean
- file.mode -> string
- file.name -> string
- file.encoding -> string


# fileinput – Process lines from input streams

- Purpose Create command-line filter programs to process lines from input streams.
- The fileinput module is a framework for creating command line programs for processing text files in a filter-ish manner

- Iterate over lines from multiple input streams


# fileinput – Process lines from input streams

- This module implements a helper class and functions to quickly write a loop over standard input or a list of files.
- The typical use is:

```
import fileinput
for line in fileinput.input():
        print line
```

- This iterates over the lines of all files listed in sys.argv[1:], defaulting to sys.stdin if the list is empty. If
- a filename is ’-’, it is also replaced by sys.stdin. To specify an alternative list of filenames, pass it as the first
- argument to input(). A single file name is also allowed.


# stat — Interpreting stat() results

- The stat module defines constants and functions for interpreting the results of os.stat(), os.fstat() and
- os.lstat() (if they exist).
- For complete details about the stat(), fstat() and lstat() calls, consult the documentation for your system.


# stat — Interpreting stat() results

```
import stat
import os
myfile_stat = os.stat(myfile)
filesize = myfile_stat[stat.ST_SIZE]
mode = myfile_stat[stat.ST_MODE]
if stat.S_ISREG(mode):
    print '%(myfile)s is a regular file '\'with %(filesize)d bytes' % vars()
```


# filecmp — File and Directory Comparisons

- The filecmp module defines functions to compare files and directories, with various optional time/correctness tradeoffs.

- Example:
- >>> import filecmp
- >>> filecmp.cmp('libundoc.tex', 'libundoc.tex')
- 1
- >>> filecmp.cmp('libundoc.tex', 'lib.tex')
- 0


# The dircmp class

- class dircmp(a, b[, ignore[, hide ] ])
- Construct a new directory comparison object, to compare the directories a and b. ignore is a list of names
- to ignore, and defaults to [’RCS’, ’CVS’, ’tags’]. hide is a list of names to hide, and defaults to
- [os.curdir, os.pardir].

- import filecmp
- filecmp.dircmp(’example/dir1’, ’example/dir2’).report()


# glob — UNIX style pathname pattern expansion

- The glob module finds all the pathnames matching a specified pattern according to the rules used by the UNIX shell.
- No tilde expansion is done, but *, ?, and character ranges expressed with [] will be correctly matched. This is done by
- using the os.listdir() and fnmatch.fnmatch() functions in concert, and not by actually invoking a subshell.
- (For tilde and shell variable expansion, use os.path.expanduser() and os.path.expandvars().)
- glob(pathname)
- Returns a possibly-empty list of path names that match pathname, which must be a string containing a path specification.
- pathname can be either absolute (like ‘/usr/src/Python-1.5/Makefile’) or relative (like ‘../../Tools/*/*.gif’),
- and can contain shell-style wildcards.


# glob — UNIX style pathname pattern expansion

- An asterisk (*) matches zero or more characters in a segment of a name. For example, dir/*.

```
import glob
for name in glob.glob('dir/*'):
    print name
```

- The pattern matches every pathname (file or directory) in the directory dir, without recursing further into subdirectories.


# zipfile

- Purpose Read and write ZIP archive files.
- The is_zipfile() function returns a boolean indicating whether or not the filename passed as an argument refers to a valid ZIP file.

**import zipfile**

**for filename in [ 'README.txt', 'example.zip',** 'bad_example.zip', 'notthere.zip' ]:

**print '%20s %s' % (filename, zipfile.is_zipfile(filename))**

- Notice that if the file does not exist at all, is_zipfile() returns False.


# tarfile – Tar archive access

- Purpose Tar archive access.
- The is_tarfile() function returns a boolean indicating whether or not the filename passed as an argument refers to a valid tar file.

```
import tarfile
for filename in [ 'README.txt', 'example.tar', 'bad_example.tar',
    'notthere.tar' ]:
    try:
        print '%20s %s' % (filename, tarfile.is_tarfile(filename))
    except IOError, err:
        print '%20s %s' % (filename, err)
```


# pickle and cPickle

- Purpose Python object serialization
- The pickle module implements an algorithm for turning an arbitrary Python object into a series of bytes. This process is also called serializing" the object.
- The byte stream representing the object can then be transmitted or stored, and later reconstructed to create a new object with the same characteristics.
- The cPickle module implements the same algorithm, in C instead of Python.
- It is many times faster than the Python implementation, but does not allow the user to subclass from Pickle.


# pickle and cPickle

- **Encoding and Decoding Data in Strings**

```
try:
        import cPickle as pickle
except:
        import pickle
import pprint
data = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print 'DATA:',
pprint.pprint(data)
data_string = pickle.dumps(data)
print 'PICKLE:', data_string
```


# pickle_unpickle.py

```
try:
    import cPickle as pickle
except:
    import pickle
import pprint
data1 = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print 'BEFORE:',
pprint.pprint(data1)
data1_string = pickle.dumps(data1)
data2 = pickle.loads(data1_string)
print 'AFTER:',
pprint.pprint(data2)
print 'SAME?:', (data1 is data2)
print 'EQUAL?:', (data1 == data2)
```


# pickle_unpickle.py

```
try:
    import cPickle as pickle
except:
    import pickle
import pprint
data1 = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print 'BEFORE:',
pprint.pprint(data1)
data1_string = pickle.dumps(data1)
data2 = pickle.loads(data1_string)
print 'AFTER:',
pprint.pprint(data2)
print 'SAME?:', (data1 is data2)
print 'EQUAL?:', (data1 == data2)
```


# shelve

- The shelve module implements persistent storage for arbitrary Python objects which can be pickled, using a dictionary-like API.

- **Creating a new Shelf**

```
import shelve
s = shelve.open('test_shelf.db')
try:
    s['key1'] = { 'int': 10, 'float':9.5, 'string':'Sample data' }
finally:
    s.close()
```


# shelve

- To access the data again, open the shelf and use it like a dictionary:

```
import shelve
s = shelve.open('test_shelf.db')
try:
    existing = s['key1']
finally:
    s.close()
print existing
```

```
$ python shelve_existing.py
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```


# json

- **json – JavaScript Object Notation Serializer**
- Encode Python objects as JSON strings, and decode JSON strings into Python objects.
- **Encoding and Decoding Simple Data Types**

```
import json
data = [ { 'a':'A', 'b':(2, 4), 'c':3.0 } ]
print 'DATA:', repr(data)
data_string = json.dumps(data)
print 'JSON:', data_string
```

Values are encoded in a manner very similar to Python's repr() output.

```
$ python json_simple_types.py
DATA: [{'a': 'A', 'c': 3.0, 'b': (2, 4)}]
JSON: [{"a": "A", "c": 3.0, "b": [2, 4]}]
```


# json

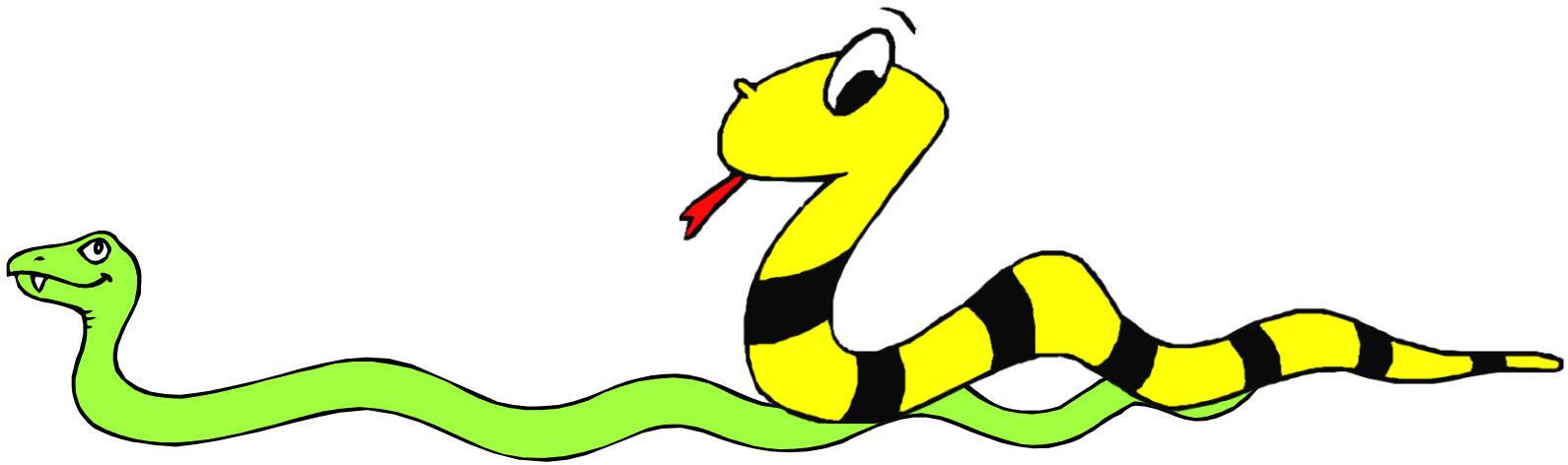
- Encoding, then re-decoding may not give exactly the same type of object.

```
import json
data = [ { 'a':'A', 'b':(2, 4), 'c':3.0 } ]
data_string = json.dumps(data)
print 'ENCODED:', data_string
decoded = json.loads(data_string)
print 'DECODED:', decoded
print 'ORIGINAL:', type(data[0]['b'])
print 'DECODED :', type(decoded[0]['b'])
```

In particular, strings are converted to unicode and tuples become lists.

```
$ python json_simple_types_decode.py
ENCODED: [{"a": "A", "c": 3.0, "b": [2, 4]}]
DECODED: [{'a': 'A', 'c': 3.0, 'b': [2, 4]}]
ORIGINAL: <type 'tuple'>
DECODED : <type 'list'>
```

# Python iterators and generators




# Iterators and generators


- Python makes good use of iterators
- And has a special kind of generator function that is powerful and useful
- We'll look at what both are
- And why they are useful
- See Norman Matloff's excellent tutorial on python iterators and generators from which some of this material is borrowed


# Files are iterators

```
>>> f = open("myfile.txt")
>>> for l in f.readlines(): print len(l)
9
21
35
43
>>> f = open("myfile.txt")
>>> for l in f: print len(l)
...
9
21
35
43
```

readlines() returns a list of the lines in file

A file is a iterator, producing new values as needed


# Files are iterators

- Iterators are supported wherever you can iterate over collections in containers (e.g., lists, tuples, dictionaries)

```
>>> f = open("myfile.txt")
>>> map(len, f.readlines())
[9, 21, 35, 43]
>>> f = open("myfile.txt")
>>> map(len, f)
[9, 21, 35, 43]
>>>
```


# Like sequences, but…

- Iterators are like sequences (lists, tuples), but…
- The entire sequence is not manifested
- Items produced one at a time when and as needed
- The sequence can be infinite (e.g., all positive integers)
- You can create your own iterators if you write a function to generate the next item


# Example: fib.py

```
class fibnum:
    def __init__(self):
        self.fn2 = 1
        self.fn1 = 1

    def next(self): # next() is the heart of any iterator
         # use of the following tuple to not only save lines of
         # code but insures that only the old values of self.fn1 and
         # self.fn2 are used in assigning the new values
        (self.fn1, self.fn2, oldfn2) = (self.fn1+self.fn2, self.fn1, self.fn2)
        return oldfn2

    def __iter__(self):
         return self
```

next() used to generate successive values

Classes with an __iter__() method are iterators


# Example: fib.py

```
>>> from fib import *
>>> f = fibnum()
>>> for i in f:
...   print i
...   if i > 100: break
1
1
2
3
…
144
>>>
```


# Stopping an iterator

```
class fibnum20:
    def __init__(self):
        self.fn2 = 1 # "f_{n-2}"
        self.fn1 = 1 # "f_{n-1}"

    def next(self):
        (self.fn1,self.fn2,oldfn2) = (self.fn1+self.fn2,self.fn1,self.fn2)
        if oldfn2 > 20: raise StopIteration
        return oldfn2

    def __iter__(self):
        return self
```

Raise this error to tell consumer to stop


# Stopping an iterator

```
>>> from fib import *
>>> for i in fibnum20(): print i
1
1
2
3
5
8
13
>>>
```


# More tricks

- The list function materializes an iterator's values as a list

```
>>> list(fibnum20())
[1, 1, 2, 3, 5, 8, 13
```

- sum(), max(), min() know about iterators

```
>>> sum(fibnum20())
33
>>> max(fibnum20())
13
>>> min(fibnum20())
1
```


# itertools

- The itertools library module has some useful tools for working with iterators
- islice() is like slice but works with streams produced by iterators

```
>>> from itertools import *
>>> list(islice(fibnum(), 6))
[1, 1, 2, 3, 5, 8]
>>> list(islice(fibnum(), 6, 10))
[13, 21, 34, 55]
```

- See also imap, ifilter, …

# Python generators


- Python generators generate iterators
- They are more powerful and convenient
- Write a regular function and instead of calling return to produce a value, call yield instead
- When another value is needed, the generator function picks up where it left off
- Raise the StopIteration exception or call return when you are done


# Generator example

```
def gy():
  x = 2
  y = 3
  yield x,y,x+y
  z = 12
  yield z/x
  yield z/y
  return
```

```
>>> from gen import *
>>> g = gy()
>>> g.next()
(2, 3, 5)
>>> g.next()
6
>>> g.next()
4
>>> g.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```


# Generator example: fib()

```
def fib( ):
    fn2 = 1
    fn1 = 1
    while True:
        (fn1,fn2,oldfn2) = (fn1+fn2,fn1,fn2)
        yield oldfn2
```


# Generator example: getword()

```
def getword(fl):
  for line in fl:
    for word in line.split():
      yield word
  return
```


# Remembers stack, too

```
def inorder(tree):
  if tree:
    for x in inorder(tree.left):
      yield x
      yield tree.dat
      for x in inorder(tree.right):
        yield x
```


# collections — High-performance container datatypes

- This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple.
- deque list-like container with fast appends and pops on either end.
- Counter dict subclass for counting hashable objects
- OrderedDict dict subclass that remembers the order entries were added.
- Defaultdict dict subclass that calls a factory function to supply missing values


# deque objects

- *class* collections.deque([*iterable*[, *maxlen*]]) Returns a new deque object initialized left-to-right (using append()) with data from *iterable*. If *iterable* is not specified, the new deque is empty.
- Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”).
- Deques support thread-safe, memory efficient appends and pops from either side of the deque.
- Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation


# deque objects

- Deque objects support the following methods:
- append(*x*) Add *x* to the right side of the deque.
- appendleft(*x*) Add *x* to the left side of the deque.
- clear() Remove all elements from the deque leaving it with length 0.
- count(*x*) Count the number of deque elements equal to *x*.
- extend(*iterable*) Extend the right side of the deque by appending elements from the iterable argument.
- extendleft(*iterable*) Extend the left side of the deque by appending elements from *iterable*.


# deque objects

- **pop()** Remove and return an element from the right side of the deque. If no elements are present, raises an IndexError.
- **popleft()** Remove and return an element from the left side of the deque. If no elements are present, raises an IndexError.
- **remove(*value*)**Removed the first occurrence of *value*. If not found, raises a ValueError.
- **reverse()**Reverse the elements of the deque in-place and then return None.
- rotate(*n*) Rotate the deque *n* steps to the right. If *n* is negative, rotate to the left. Rotating one step to the right is equivalent to: d.appendleft(d.pop()).
- Deque objects also provide one read-only attribute:
- maxlen Maximum size of a deque or *None* if unbounded.


# deque objects

```
>>> from collections import deque
 >>> d = deque('ghi') # make a new deque with three
    items
>>> for elem in d: # iterate over the deque's elements
    ... print elem.upper()
 G H I
>>> d.append('j') # add a new entry to the right side
>>> d.appendleft('f') # add a new entry to the left side
>>> d # show the representation of the deque
    deque(['f', 'g', 'h', 'i', 'j'])
```


# deque objects

```
>>> d.pop() # return and remove the rightmost item
'j'
 >>> d.popleft() # return and remove the leftmost item 'f'
>>> list(d) # list the contents of the deque
['g', 'h', 'i']
>>> d[0] # peek at leftmost item
 'g'
>>> d[-1] # peek at rightmost item
 'i'
>>> list(reversed(d)) # list the contents of a deque in reverse
 ['i', 'h', 'g']
>>> 'h' in d # search the deque
True
```


# deque objects

```
>>> d.extend('jkl') # add multiple elements at once
 >>> d
 deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1) # right rotation
 >>> d
 deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1) # left rotation
 >>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
```


# deque objects

```
>>> deque(reversed(d)) # make a new deque in reverse order
    deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear() # empty the deque
>>> d.pop() # cannot pop from an empty deque
Traceback (most recent call last):
File "<pyshell#6>", line 1, in -toplevel- d.pop()
IndexError: pop from an empty deque
 >>> d.extendleft('abc') # extendleft() reverses the input order
 >>> d
deque(['c', 'b', 'a'])
```


# defaultdict objects

- *class* collections.defaultdict([*default_factory*[, ...]])
- Returns a new dictionary-like object.
- defaultdict is a subclass of the built-in dict class.
- It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the dict class and is not documented here.
- The first argument provides the initial value for the default_factory attribute; it defaults to None.
- All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.


# defaultdict Examples

- Using list as the default_factory, it is easy to group a sequence of key-value pairs into a dictionary of lists:
- **>>>** s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
- **>>>** d = defaultdict(list)
- **>>> for** k, v **in** s:
- **...** d[k].append(v)
- **...**
- **>>>** d.items()

[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]


# defaultdict Examples

- **>>>** d = {}
- **>>> for** k, v **in** s:
- **...** d.setdefault(k, []).append(v)
- **...**
- **>>>** d.items()
- [('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]


# defaultdict Examples

- Setting the default_factory to int makes the defaultdict useful for counting (like a bag or multiset in other languages):
- **>>>** s = 'mississippi'
- **>>>** d = defaultdict(int)
- **>>> for** k **in** s:
- **...** d[k] += 1
- **...**
- **>>>** d.items()
- [('i', 4), ('p', 2), ('s', 4), ('m', 1)]


# Contact Us on:

**G K T C S Innovations Pvt. Ltd.**

**IT Training, Consultancy, Software Development, Staffing**
**#11,4th Floor,Sneh Deep, Near Warje Flyover Bridge,**
**Warje-Malwadi, Pune -411058, Maharashtra, India.**
**Mobile: +91- 9975072320, 8308761477**
**Email : surendra@gktcs.com**
**Web: www.gktcs.com**