Pattern Matching

Pattern Matching

- using the `re` module
- matching strings with match()

Regular Expressions

- common regular expression symbols
- groups of regular expressions

The Meaning of Python

- a dynamic language
- rapid application development

MCS 275 Lecture 41
Programming Tools and File Management
Jan Verschelde, 25 April 2008

Pattern Matching

Regular Expressions

using the `re` module

Manipulating text and strings is an important task:

- parse to ensure entered data is correct,
- search through confidential data: use program.

Suppose `answer` contains the answers to a yes or no question.

Acceptable *yes* answers:

1. `y` or `yes`
2. `Y` or `Yes`

Testing all these cases is tedious.

Support for **r**egular **e**xpressions:

```
>>> import re
```

`re` is a standard library module.

Regular Expressions

using the `re` module

Manipulating text and strings is an important task:

- parse to ensure entered data is correct,
- search through confidential data: use program.

Suppose `answer` contains the answers to a yes or no question.

Acceptable *yes* answers:

1. `y` or `yes`
2. `Y` or `Yes`

Testing all these cases is tedious.

Support for **r**egular **e**xpressions:

```
>>> import re
```

`re` is a standard library module.

Regular Expressions

using the `re` module

Manipulating text and strings is an important task:

- parse to ensure entered data is correct,
- search through confidential data: use program.

Suppose `answer` contains the answers
to a yes or no question.

Acceptable *yes* answers:

1. `y` or `yes`
2. `Y` or `Yes`

Testing all these cases is tedious.

Support for **r**egular **e**xpressions:

```
>>> import re
```

`re` is a standard library module.

Regular Expressions

using the `re` module

Manipulating text and strings is an important task:

- parse to ensure entered data is correct,
- search through confidential data: use program.

Suppose `answer` contains the answers
to a yes or no question.

Acceptable *yes* answers:

1. `y` or `yes`
2. `Y` or `Yes`

Testing all these cases is tedious.

Support for **r**egular **e**xpressions:

```
>>> import re
```

`re` is a standard library module.

Regular Expressions

using the `re` module

Manipulating text and strings is an important task:

- parse to ensure entered data is correct,
- search through confidential data: use program.

Suppose `answer` contains the answers
to a yes or no question.

Acceptable *yes* answers:

1. `y` or `yes`
2. `Y` or `Yes`

Testing all these cases is tedious.

Support for **r**egular **e**xpressions:

```
>>> import re
```

`re` is a standard library module.

Matching short and long Answers

using re module

```
>>> import re
>>> short = 'y'; long = 'Yes'
>>> re.match('y',short) != None
True
>>> re.match('y',long) != None
False
>>> re.match('y|Y',long) != None
True
>>> re.match('y|Y',long)
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('y|Y',long).group()
'Y'
>>> re.match('y|Y',short).group()
'y'
```

Matching short and long Answers

using re module

```
>>> import re
>>> short = 'y'; long = 'Yes'
>>> re.match('y',short) != None
True
>>> re.match('y',long) != None
False
>>> re.match('y|Y',long) != None
True
>>> re.match('y|Y',long)
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('y|Y',long).group()
'Y'
>>> re.match('y|Y',short).group()
'y'
```

Matching short and long Answers

using re module

```
>>> import re
>>> short = 'y'; long = 'Yes'
>>> re.match('y',short) != None
True
>>> re.match('y',long) != None
False
>>> re.match('y|Y',long) != None
True
>>> re.match('y|Y',long)
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('y|Y',long).group()
'Y'
>>> re.match('y|Y',short).group()
'y'
```

Matching short and long Answers

using re module

```
>>> import re
>>> short = 'y'; long = 'Yes'
>>> re.match('y',short) != None
True
>>> re.match('y',long) != None
False
>>> re.match('y|Y',long) != None
True
>>> re.match('y|Y',long)
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('y|Y',long).group()
'Y'
>>> re.match('y|Y',short).group()
'y'
```

Matching short and long Answers

using re module

```
>>> import re
>>> short = 'y'; long = 'Yes'
>>> re.match('y',short) != None
True
>>> re.match('y',long) != None
False
>>> re.match('y|Y',long) != None
True
>>> re.match('y|Y',long)
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('y|Y',long).group()
'Y'
>>> re.match('y|Y',short).group()
'y'
```

Matching short and long Answers

using re module

```
>>> import re
>>> short = 'y'; long = 'Yes'
>>> re.match('y',short) != None
True
>>> re.match('y',long) != None
False
>>> re.match('y|Y',long) != None
True
>>> re.match('y|Y',long)
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('y|Y',long).group()
'Y'
>>> re.match('y|Y',short).group()
'y'
```

Matching short and long Answers

using re module

```
>>> import re
>>> short = 'y'; long = 'Yes'
>>> re.match('y',short) != None
True
>>> re.match('y',long) != None
False
>>> re.match('y|Y',long) != None
True
>>> re.match('y|Y',long)
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('y|Y',long).group()
'Y'
>>> re.match('y|Y',short).group()
'y'
```

Pattern Matching

Matching Strings with `match()`

creating match objects

The function `match()` in the `re` module:

```
>>> re.match( < pattern > , < string > )
```

If the `string` does not match the `pattern`,
then `None` is returned.

If the `string` matches the `pattern`,
then a match object is returned.

```
>>> re.match('he','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('hi','hello') == None
True
```

Matching Strings with `match()`

creating match objects

The function `match()` in the `re` module:

```
>>> re.match( < pattern > , < string > )
```

If the `string` does not match the `pattern`,
then `None` is returned.

If the `string` matches the `pattern`,
then a match object is returned.

```
>>> re.match('he','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('hi','hello') == None
True
```

Matching Strings with `match()`

creating match objects

The function `match()` in the `re` module:

```
>>> re.match( < pattern > , < string > )
```

If the `string` does not match the `pattern`,
then `None` is returned.

If the `string` matches the `pattern`,
then a match object is returned.

```
>>> re.match('he','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('hi','hello') == None
True
```

Matching Strings with `match()`

creating match objects

The function `match()` in the `re` module:

```
>>> re.match( < pattern > , < string > )
```

If the `string` does not match the `pattern`,
then `None` is returned.

If the `string` matches the `pattern`,
then a match object is returned.

```
>>> re.match('he','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('hi','hello') == None
True
```

Matching Strings with `match()`

creating match objects

The function `match()` in the `re` module:

```
>>> re.match( < pattern > , < string > )
```

If the `string` does not match the `pattern`,
then `None` is returned.

If the `string` matches the `pattern`,
then a match object is returned.

```
>>> re.match('he','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> re.match('hi','hello') == None
True
```

The group() Method

using a match object

What can we do with the match object?

```
>>> re.match('he','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> _.group()
'he'
```

After a successful match, group() returns that part of the pattern that matches the string.

The group() Method

using a match object

What can we do with the match object?

```
>>> re.match('he','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> _.group()
'he'
```

After a successful match, `group()` returns that part of the pattern that matches the string.

Searching versus Matching

the methods `search()` and `match()`

The match only works from the start:

```
>>> re.match('ell','hello') == None
True
```

Looking for the first occurrence of the pattern in the string, with `search()`:

```
>>> re.search('ell','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> _.group()
'ell'
```

Searching versus Matching

the methods `search()` and `match()`

The match only works from the start:

```
>>> re.match('ell','hello') == None
True
```

Looking for the first occurrence of the pattern in the string, with `search()`:

```
>>> re.search('ell','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> _.group()
'ell'
```

Searching versus Matching

the methods `search()` and `match()`

The match only works from the start:

```
>>> re.match('ell','hello') == None
True
```

Looking for the first occurrence of the pattern in the string, with `search()`:

```
>>> re.search('ell','hello')
<_sre.SRE_Match object at 0x5cb10>
>>> _.group()
'ell'
```

Pattern Matching

Regular Expressions

common regular expression symbols

pattern	strings matched
`literal`	strings starting with `literal`
`re1|re2`	strings starting with `re1` or `re2`

```
>>> from time import ctime
>>> now = ctime()
>>> now
'Fri Apr 25 07:32:07 2008'
>>> p = '\w{3}\s\w{3}\s\d{2}\s\d{2}:\d{2}:\d{2}\s\d{4}'
>>> re.match(p,now) != None
True
```

pattern	strings matched
`\w`	any alphanumeric character, same as `[A-Za-z]`
`\d`	any decimal digit, same as `[0-9]`
`\s`	any whitespace character
`re{`*n*`}`	*n* occurrences of `re`

Regular Expressions

common regular expression symbols

pattern	strings matched
`literal`	strings starting with `literal`
`re1|re2`	strings starting with `re1` or `re2`

```
>>> from time import ctime
>>> now = ctime()
>>> now
'Fri Apr 25 07:32:07 2008'
>>> p = '\w{3}\s\w{3}\s\d{2}\s\d{2}:\d{2}:\d{2}\s\d{4}'
>>> re.match(p,now) != None
True
```

pattern	strings matched
`\w`	any alphanumeric character, same as `[A-Za-z]`
`\d`	any decimal digit, same as `[0-9]`
`\s`	any whitespace character
`re{`*n*`}`	*n* occurrences of `re`

Regular Expressions

common regular expression symbols

pattern	strings matched
`literal`	strings starting with `literal`
`re1|re2`	strings starting with `re1` or `re2`

```
>>> from time import ctime
>>> now = ctime()
>>> now
'Fri Apr 25 07:32:07 2008'
>>> p = '\w{3}\s\w{3}\s\d{2}\s\d{2}:\d{2}:\d{2}\s\d{4}'
>>> re.match(p,now) != None
True
```

pattern	strings matched
`\w`	any alphanumeric character, same as `[A-Za-z]`
`\d`	any decimal digit, same as `[0-9]`
`\s`	any whitespace character
`re{`*n*`}`	*n* occurrences of `re`

Regular Expressions

common regular expression symbols

pattern	strings matched
`literal`	strings starting with `literal`
`re1|re2`	strings starting with `re1` or `re2`

```
>>> from time import ctime
>>> now = ctime()
>>> now
'Fri Apr 25 07:32:07 2008'
>>> p = '\w{3}\s\w{3}\s\d{2}\s\d{2}:\d{2}:\d{2}\s\d{4}'
>>> re.match(p,now) != None
True
```

pattern	strings matched
`\w`	any alphanumeric character, same as `[A-Za-z]`
`\d`	any decimal digit, same as `[0-9]`
`\s`	any whitespace character
`re{`*n*`}`	*n* occurrences of `re`

Matching 0 or 1 Occurrences

allow `Ms.`, `Mr.`, `Mrs.`, with or without the `.` (dot)

```
>>> title = 'Mr?s?\.? '
```

- `?` matches 0 or 1 occurrences
- `.` matches any character
- `\.` matches the dot `.`

```
>>> re.match(title,'Ms ') != None
True
>>> re.match(title,'Ms. ') != None
True
>>> re.match(title,'Miss ') != None
False
>>> re.match(title,'Mr') != None
False
>>> re.match(title,'Mr ') != None
True
>>> re.match(title,'M ') != None
True
```

Matching 0 or 1 Occurrences

allow `Ms.`, `Mr.`, `Mrs.`, with or without the `.` (dot)

```
>>> title = 'Mr?s?\.? '
```

- `?` matches 0 or 1 occurrences
- `.` matches any character
- `\.` matches the dot `.`

```
>>> re.match(title,'Ms ') != None
True
>>> re.match(title,'Ms. ') != None
True
>>> re.match(title,'Miss ') != None
False
>>> re.match(title,'Mr') != None
False
>>> re.match(title,'Mr ') != None
True
>>> re.match(title,'M ') != None
True
```

Matching 0 or 1 Occurrences

allow `Ms.`, `Mr.`, `Mrs.`, with or without the `.` (dot)

```
>>> title = 'Mr?s?\.? '
```

- `?` matches 0 or 1 occurrences
- `.` matches any character
- `\.` matches the dot `.`

```
>>> re.match(title,'Ms ') != None
True
>>> re.match(title,'Ms. ') != None
True
>>> re.match(title,'Miss ') != None
False
>>> re.match(title,'Mr') != None
False
>>> re.match(title,'Mr ') != None
True
>>> re.match(title,'M ') != None
True
```

Matching 0 or 1 Occurrences

allow `Ms.`, `Mr.`, `Mrs.`, with or without the `.` (dot)

```
>>> title = 'Mr?s?\.? '
```

- `?` matches 0 or 1 occurrences
- `.` matches any character
- `\.` matches the dot `.`

```
>>> re.match(title,'Ms ') != None
True
>>> re.match(title,'Ms. ') != None
True
>>> re.match(title,'Miss ') != None
False
>>> re.match(title,'Mr') != None
False
>>> re.match(title,'Mr ') != None
True
>>> re.match(title,'M ') != None
True
```

Matching 0 or 1 Occurrences

allow `Ms.`, `Mr.`, `Mrs.`, with or without the `.` (dot)

```
>>> title = 'Mr?s?\.? '
```

- `?` matches 0 or 1 occurrences
- `.` matches any character
- `\.` matches the dot `.`

```
>>> re.match(title,'Ms ') != None
True
>>> re.match(title,'Ms. ') != None
True
>>> re.match(title,'Miss ') != None
False
>>> re.match(title,'Mr') != None
False
>>> re.match(title,'Mr ') != None
True
>>> re.match(title,'M ') != None
True
```

Matching 0 or 1 Occurrences

allow `Ms.`, `Mr.`, `Mrs.`, with or without the `.` (dot)

```
>>> title = 'Mr?s?\.? '
```

- `?` matches 0 or 1 occurrences
- `.` matches any character
- `\.` matches the dot `.`

```
>>> re.match(title,'Ms ') != None
True
>>> re.match(title,'Ms. ') != None
True
>>> re.match(title,'Miss ') != None
False
>>> re.match(title,'Mr') != None
False
>>> re.match(title,'Mr ') != None
True
>>> re.match(title,'M ') != None
True
```

Character Classes

Match with specific characters.

A name has to start with upper case:

```
>>> name = '[A-Z][a-z]*'
>>> G = 'Guido van Rossum'
>>> re.match(name,G)
>>> _.group()
'Guido'

>>> g = 'guido'
>>> re.match(name,g) == None
True
```

Character Classes

Match with specific characters.

A name has to start with upper case:

```
>>> name = '[A-Z][a-z]*'
>>> G = 'Guido van Rossum'
>>> re.match(name,G)
>>> _.group()
'Guido'

>>> g = 'guido'
>>> re.match(name,g) == None
True
```

Character Classes

Match with specific characters.

A name has to start with upper case:

```
>>> name = '[A-Z][a-z]*'
>>> G = 'Guido van Rossum'
>>> re.match(name,G)
>>> _.group()
'Guido'

>>> g = 'guido'
>>> re.match(name,g) == None
True
```

Pattern Matching

the groups() method

Groups of regular expressions are designated with parenthesis, between (and).

Syntax:

```
< pattern > = ( < group1 > ) ( < group2 > )
m = re.match( < pattern > , < string > )
if m != None: m.groups()
```

After a successful match, groups() returns a tuple of those parts of the string that matched the pattern.

the groups() method

Groups of regular expressions are designated with parenthesis, between (and).

Syntax:

```
< pattern > = ( < group1 > ) ( < group2 > )
m = re.match( < pattern > , < string > )
if m != None: m.groups()
```

After a successful match, groups() returns a tuple of those parts of the string that matched the pattern.

the `groups()` method

Groups of regular expressions are designated with parenthesis, between `(` and `)`.

Syntax:

```
< pattern > = ( < group1 > ) ( < group2 > )
m = re.match( < pattern > , < string > )
if m != None: m.groups()
```

After a successful match, groups() returns a tuple of those parts of the string that matched the pattern.

Extracting hours, seconds, minutes

using the `groups()` method

```
>>> from time import ctime
>>> now = ctime()
>>> now
'Fri Apr 25 07:32:07 2008'
>>> t = now.split(' ')[3]
>>> t
'07:32:07'
>>> format = '(\d\d):(\d\d):(\d\d)'
>>> m = re.match(format,t)
>>> m.groups()
('07', '32', '07')
>>> (hours, minutes, seconds) = _
>>> minutes
'32'
```

Extracting hours, seconds, minutes

using the `groups()` method

```
>>> from time import ctime
>>> now = ctime()
>>> now
'Fri Apr 25 07:32:07 2008'
>>> t = now.split(' ')[3]
>>> t
'07:32:07'
>>> format = '(\d\d):(\d\d):(\d\d)'
>>> m = re.match(format,t)
>>> m.groups()
('07', '32', '07')
>>> (hours, minutes, seconds) = _
>>> minutes
'32'
```

Extracting hours, seconds, minutes

using the `groups()` method

```
>>> from time import ctime
>>> now = ctime()
>>> now
'Fri Apr 25 07:32:07 2008'
>>> t = now.split(' ')[3]
>>> t
'07:32:07'
>>> format = '(\d\d):(\d\d):(\d\d)'
>>> m = re.match(format,t)
>>> m.groups()
('07', '32', '07')
>>> (hours, minutes, seconds) = _
>>> minutes
'32'
```

Extracting hours, seconds, minutes

using the `groups()` method

```
>>> from time import ctime
>>> now = ctime()
>>> now
'Fri Apr 25 07:32:07 2008'
>>> t = now.split(' ')[3]
>>> t
'07:32:07'
>>> format = '(\d\d):(\d\d):(\d\d)'
>>> m = re.match(format,t)
>>> m.groups()
('07', '32', '07')
>>> (hours, minutes, seconds) = _
>>> minutes
'32'
```

Extracting hours, seconds, minutes

using the `groups()` method

```
>>> from time import ctime
>>> now = ctime()
>>> now
'Fri Apr 25 07:32:07 2008'
>>> t = now.split(' ')[3]
>>> t
'07:32:07'
>>> format = '(\d\d):(\d\d):(\d\d)'
>>> m = re.match(format,t)
>>> m.groups()
('07', '32', '07')
>>> (hours, minutes, seconds) = _
>>> minutes
'32'
```

Pattern Matching

The Development Cycle

compilation versus interpretation

Traditional build cycle:

1. run the application
2. test the behavior of the code
3. stop the application
4. edit the program code
5. recompile the code
6. relink executable
7. goto step 1

This is the static language build cycle.

Python eliminates steps 4 and 5.

Recompilation and relinking is not a trivial task for systems with over 100,000 lines of code.

The Development Cycle

compilation versus interpretation

Traditional build cycle:

1. run the application
2. test the behavior of the code
3. stop the application
4. edit the program code
5. recompile the code
6. relink executable
7. goto step 1

This is the static language build cycle.

Python eliminates steps 4 and 5.

Recompilation and relinking is not a trivial task for systems with over 100,000 lines of code.

The Development Cycle

compilation versus interpretation

Traditional build cycle:

1. run the application
2. test the behavior of the code
3. stop the application
4. edit the program code
5. recompile the code
6. relink executable
7. goto step 1

This is the static language build cycle.

Python eliminates steps 4 and 5.

Recompilation and relinking is not a trivial task for systems with over 100,000 lines of code.

The Development Cycle

compilation versus interpretation

Traditional build cycle:

1. run the application
2. test the behavior of the code
3. stop the application
4. edit the program code
5. recompile the code
6. relink executable
7. goto step 1

This is the static language build cycle.

Python eliminates steps 4 and 5.

Recompilation and relinking is not a trivial task for systems with over 100,000 lines of code.

The Development Cycle

compilation versus interpretation

Traditional build cycle:

1. run the application
2. test the behavior of the code
3. stop the application
4. edit the program code
5. recompile the code
6. relink executable
7. goto step 1

This is the static language build cycle.

Python eliminates steps 4 and 5.

Recompilation and relinking is not a trivial task for systems with over 100,000 lines of code.

The Development Cycle

compilation versus interpretation

Traditional build cycle:

1. run the application
2. test the behavior of the code
3. stop the application
4. edit the program code
5. recompile the code
6. relink executable
7. goto step 1

This is the static language build cycle.

Python eliminates steps 4 and 5.

Recompilation and relinking is not a trivial task for systems with over 100,000 lines of code.

The Development Cycle

compilation versus interpretation

Traditional build cycle:

1. run the application
2. test the behavior of the code
3. stop the application
4. edit the program code
5. recompile the code
6. relink executable
7. goto step 1

This is the static language build cycle.

Python eliminates steps 4 and 5.

Recompilation and relinking is not a trivial task for systems with over 100,000 lines of code.

The Development Cycle

compilation versus interpretation

Traditional build cycle:

1. run the application
2. test the behavior of the code
3. stop the application
4. edit the program code
5. recompile the code
6. relink executable
7. goto step 1

This is the static language build cycle.

Python eliminates steps 4 and 5.

Recompilation and relinking is not a trivial task for systems with over 100,000 lines of code.

The Development Cycle

compilation versus interpretation

Traditional build cycle:

1. run the application
2. test the behavior of the code
3. stop the application
4. edit the program code
5. recompile the code
6. relink executable
7. goto step 1

This is the static language build cycle.

Python eliminates steps 4 and 5.

Recompilation and relinking is not a trivial task for systems with over 100,000 lines of code.

Python is a dynamic Language

some advertisements

Python may be a scripting language,
but it scales well.

- "Python is executable pseudocode."
  very readable,
  complexities involving memory addresses are hidden
- "Python is OOP done right."
  compared to C++,
  the class mechanism in Python is much simpler

The statements are quotes taken from Mark Lutz:
Programming Python book. 2nd edition, O'Reilly 2001.

Python is a dynamic Language

some advertisements

Python may be a scripting language,
but it scales well.

- “Python is executable pseudocode."
  very readable,
  complexities involving memory addresses are hidden
- “Python is OOP done right."
  compared to C++,
  the class mechanism in Python is much simpler

The statements are quotes taken from Mark Lutz:
Programming Python book. 2nd edition, O’Reilly 2001.

Python is a dynamic Language

some advertisements

Python may be a scripting language,
but it scales well.

- “Python is executable pseudocode."
  very readable,
  complexities involving memory addresses are hidden
- “Python is OOP done right."
  compared to C++,
  the class mechanism in Python is much simpler

The statements are quotes taken from Mark Lutz:
Programming Python book. 2nd edition, O’Reilly 2001.

Pattern Matching

Rapid Application Development

an answer to the software crisis

prototyping:	all Python
hybrid:	mixture of Python and C/C++
delivery:	all C/C++

Consider the slider:

prototyping	hybrid	delivery
all Python	mixture	all C/C++

Identifying the bottlenecks in a working prototype leads to a gradual development of modules in C/C++.

Python complements languages like C/C++ and Java.

Rapid Application Development

an answer to the software crisis

prototyping:	all Python
hybrid:	mixture of Python and C/C++
delivery:	all C/C++

Consider the slider:

prototyping	hybrid	delivery
all Python	mixture	all C/C++

Identifying the bottlenecks in a working prototype leads to a gradual development of modules in C/C++.

Python complements languages like C/C++ and Java.

Rapid Application Development

an answer to the software crisis

prototyping:	all Python
hybrid:	mixture of Python and C/C++
delivery:	all C/C++

Consider the slider:

prototyping	hybrid	delivery
all Python	mixture	all C/C++

Identifying the bottlenecks in a working prototype leads to a gradual development of modules in C/C++.

Python complements languages like C/C++ and Java.

Sinking the Titanic

the iceberg that sank so many large software projects...

The Titanic refers to the iceberg. What we see is the exposed interface (the domain of Python), what is hidden are the system internals (the domain of C/C++).

"Python provides a simple but powerful rapid development language, *along with the integration tools to apply it in realistic development environments."*

Mark Lutz, page 1195

The idea of integration is not new,
but the innovation is in providing the tools.

Python is open source and cross platform.

Sinking the Titanic

the iceberg that sank so many large software projects...

The Titanic refers to the iceberg. What we see is the exposed interface (the domain of Python), what is hidden are the system internals (the domain of C/C++).

"Python provides a simple but powerful rapid development language, *along with the integration tools to apply it in realistic development environments."*

Mark Lutz, page 1195

The idea of integration is not new,
but the innovation is in providing the tools.

Python is open source and cross platform.

Sinking the Titanic

the iceberg that sank so many large software projects...

The Titanic refers to the iceberg. What we see is the exposed interface (the domain of Python), what is hidden are the system internals (the domain of C/C++).

"Python provides a simple but powerful rapid development language, *along with the integration tools to apply it in realistic development environments."*

Mark Lutz, page 1195

The idea of integration is not new,
but the innovation is in providing the tools.

Python is open source and cross platform.

Summary + Assignments

We ended with yet another module, useful for parsing. Assignments:

1. Phone numbers may start with a three digit area code followed by seven numbers. Give a pattern to accept 7 or 10 digit numbers. Group the pattern so the area code may be extracted with `groups()`.
2. A safe password should contain at least one digit and at least one alphanumeric character. The digit and the alphanumeric character may occur anywhere in the password. Write a pattern for this requirement.