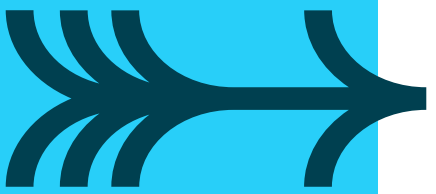
POWERING POTENTIAL

Python 3 Programming

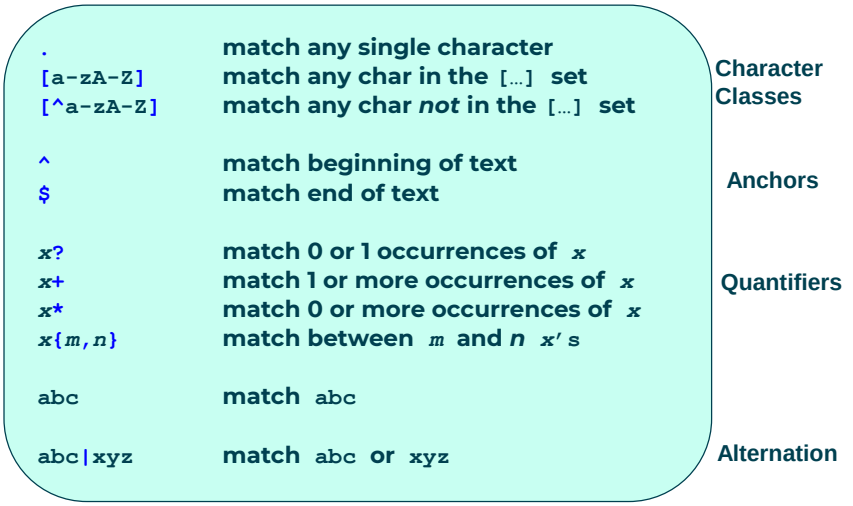
Appendix
Advanced Regular Expressions

ADVANCED REGULAR EXPRESSIONS



- Review
- Regular expression quoting
- Modifiers
- Side effect variables and capturing
- Minimal matches
- Multi-line matching
- Global matches
- Look-around assertions
- Substitution with interpolation
- Substitution with expressions

Elementary extended RE meta-characters



The examples listed above, cover the most common regular expression meta-characters. If you are new to regular expressions, this is a good table to remember.
The expression Character Class is introduced here. A Character Class may be specified in the 'traditional' way as show, using the square brackets [ ].
Note that meta-characters used inside [...] are different to those used outside. Escaped meta-characters (those prefixed with /) are literals, as are meta-characters inside [].

Regular expression objects

- **`import re` to use them**
- `compile` compiles the RE for efficiency, returns an re object
- **We can search or match**
- `search` searches for a pattern - like conventional RE's
- `match` matches from the start of the string
- `fullmatch` matches from the start to the end of the string (3.4)
- **All return a MatchObject, or None (False)**

```
testy = 'The quick brown fox jumps over the lazy dog'

m = re.search(r"(quick|slow).*(fox|camel)", testy)
if m:
    print('Matched', m.groups())
    print('Starting at', m.start())
    print('Ending at', m.end())
```

```
Matched ('quick', 'fox')
Starting at 4
Ending at 19
```

Importing the re module allows methods on the re class to be called, with search and match being the most common. The fullmatch method was introduced at Python 3.4.0. They all return an object of class MatchObject, on which several methods may be called.

The group in parentheses is also known as a capturing parentheses group. Text inside parentheses may be referred to later in the RE as a back-reference. A useful method is groups(), which returns a tuple containing the matched text, and may be used like back-references. Other methods include start() and end(), which return the positions of the match, and group() which returns the matched string. There are several MatchObject attributes, including re, which gives the original regular expression, and string which gives the original input string

If the search (or match) failed to find the pattern then the empty object None is returned, which is false in Boolean context.

Python RE syntax is similar to that used by lower level language libraries, such as the GNU C regex package.

We can compile our REs for efficiency, for example:

```
reobj = re.compile (r"([Ii]).*(\1)")
for line in file:
    m = reobj.match(line)
    if m:
        print m.string[m.start():m.end()]
```

Notice the use of raw strings (r"..."), these mean we do not have to escape (\) special characters like brackets and braces – particularly

useful with Regular Expressions.

Regular expression substitution

- sub returns the changed string

  re.sub(*pattern*, *replacement*, *string*[, *count*, *flags*])

- subn returns a tuple: (*changed string*, *number of changes*)

  re.subn(*pattern*, *replacement*, *string*[, *count*, *flags*])

- ***count* gives the number of replacements**
- Default is to replace *all* occurrences

```
line = 'Perl for Perl Programmers'
txt = re.subn('Perl', 'Python', line)
if txt[1]:
    print(txt[0])
```

```
Python for Python Programmers
```

```
txt = re.subn('Perl', 'Python', line, 1)
if txt[1]:
    print(txt[0])
```

```
Python for Perl Programmers
```

Substitution is reminiscent of awk, in that we have a couple of method calls. sub is used where we just want the new string, whereas subn returns both the altered string and the number of matches. Both perform global substitutions from the left of the string.
There are other useful re functions, for example split, which is shown on the next slide.
We have also shown a compiled Regular Expression object. Compiling the RE, makes for more efficient code when the same pattern is used many times. For example, in a loop. Methods like match, fullmatch, findall, search, split, sub, and subn may be called on these objects.

Compiled REs and RE objects

- **We can compile for efficiency**

```
reobj = re.compile (r"(I).*(\1)")
for line in file:
    m = reobj.match(line)
    if m:
        print(m.string[m.start():m.end()])
```

- **We can also compile to obtain an re object**
- Method call parameters are different!

We have shown a compiled Regular Expression object. Compiling the RE makes for more efficient code when the same pattern is used many times, for example in a loop. Methods like match, findall, search, split, sub, and subn may be called on these objects, but the argument lists may be different to the versions you are used to.

Flags

- **Change the behaviour of the match**

Long name	Short	RE	
re.ASCII	re.A	(?a)	Class shortcuts do not include Unicode
re.IGNORECASE	re.I	(?i)	Case insensitive match
re.LOCALE	re.L	(?L)	Class shortcuts are locale sensitive
re.MULTILINE	re.M	(?m)	^ and $ match start and end of *line*
re.DOTALL	re.S	(?s)	. also matches a new-line
re.VERBOSE	re.X	(?x)	Whitespace is ignored, allow comments

- May be embedded in the RE
- May be specified as an optional argument to
- re.search, re.match, and re.compile
- Multiple flags may be combined

```
m = re.search(r'(?im)^john', name)
m = re.search(r'^john', name, re.IGNORECASE|re.MULTILINE)
```

In other products, like sed and Perl 5, flags are placed after the regular expression, but in Python they are set before (Perl 6 uses a similar syntax). It would be tempting to state that the short names are the initial letter of the long name, and that the RE syntax is just the short name in lowercase. You can see that this is not the case, the S and X flags are there for compatibility with other RE engines (e.g. Perl).
The examples combine the IGNORECASE and MULTILINE flags. They look for 'john' in any case at the start of the text *or* immediately after a new-line character.
When embedded in the RE, the single characters can be in any order. When using the re module attribute flags, they are combined with a binary OR |, also in any order. The flags and embedded attributes may be mixed, but that might make the RE even more confusing.

Global matches

re.findall

→ Returns a list of matches or groups

```
str='/dev/sd3d 135398 69683 52176 57% /home/stuff'
nums = re.findall(r'\b\d+\b', str)
print(nums)
```

```
['135398', '69683', '52176', '57']
```

re.finditer

→ Returns an iterator to a match object

```
str='/dev/sd3d 135398 69683 52176 57% /home/stuff'

for num in re.finditer(r'\b(\d+)\b', str):
    print(num.groups())
```

```
('135398',)
('69683',)
('52176',)
('57',)
```

By default, most re methods search for the first (leftmost) occurrence of a pattern. These methods work on all occurrences and return either a list or an iterator to the occurrences. These are particularly useful for repeated patterns over multiple lines.
The `findall` method was added in Python 2.2.

Back-references

- **Python also allows 'back-references'**
- To create self-referencing regular expressions
- **Indicated by \n, representing the 'nth' set of parentheses**
- **Alternatively indicated by \g<n>**
- Python extension
- Also supports \g<0> which is the text of the whole match
- **May be used in the replacement in sub and subn**

```
str='copyright 2005-2006'
print(re.sub(r'((19|20)[0-9]{2})-((19|20)[0-9]{2})',
             r'\1-2013', str))
```

```
copyright 2005-2013
```

Python supports the use of parentheses to group a number of characters or regular expressions into a single unit and then apply a regular expression quantifier to the entire group. This can be useful when the pattern consists of recurring blocks of text or words.
The group in parentheses is also known as a capturing parentheses group. Text inside parentheses may be referred to later in the RE as a back-reference. This is done by the use of r'\1', r'\2', etc. to refer to the contents of the first and second sets of parentheses. If you nest parentheses, the order of the opening parenthesis is the order in which the back-references are allocated. Make sure you use 'raw' strings for the back-references, since '\1' is itself a special character.
The alternative back-reference syntax using r'\g<1>', r'\g<2>', etc. is a Python extension, and is associated with named captures (see the Advanced Regular Expressions appendix). It has the additional feature of supporting group 0 (zero) in r'\g<0>', which represents the whole of the matched string (r'\0' is not supported).
The use of many back-references in a RE will cause Python to work harder and increase the time taken to find a match. Use sparingly!

Non-capturing groups

- **A parenthesis group creates group values**
- Incurs some run-time overhead for keeping track of this

```
drink = 'A bottle of Miller'
pattern = 'A (glass|bottle|barrel) of (Bud|Miller|Coors)'
m = re.search(pattern, drink)
if m: print(m.groups())
```

```
('bottle', 'Miller')
```

- **`(?:)` parenthesis group without back-references**
- Incurs lower run-time overhead

```
drink = 'A bottle of Miller'
pattern = 'A (?:glass|bottle|barrel) of (?:Bud|Miller|Coors)'
m = re.search(pattern, drink)
if m: print(m.groups())
```

```
()
```

Using (?:pattern) instead of (pattern) is useful for performance tuning, or if you have to introduce an extra group in the middle of a huge expression and you don't want to update all uses of m.groups() following it.

Other match methods

- **`start()`: list of starting offsets of each match**
  - First element gives the offset to the start of the first match
- **`end()` : list of ending offsets+1 of each match**
  - First element gives the offset+1 to the end of the first match
- **lastindex : index of last group matched**

```
#      0123456789012345678901234567890123456789012345
txt = '2 456 first 3456 second third 98765 fourth 123'
m = re.search(r'(\d+) ([a-z]+) (\d+)',txt)
if m:
   print(m.groups())
   print([m.start(i) for i in range(1, m.lastindex+1)])
   print([m.end(i)   for i in range(1, m.lastindex+1)])
```

```
('456', 'first', '3456')
[2, 6, 12]
[5, 11, 16]
```

These lists are occasionally useful, offering lookups to text matching parentheses groups.

Named captures

- **Give names to captures, not just boring `m.group()`...**

```
(?P<name>RE-pattern)
```

- **Capture names and values are in `m.groupdict()`**

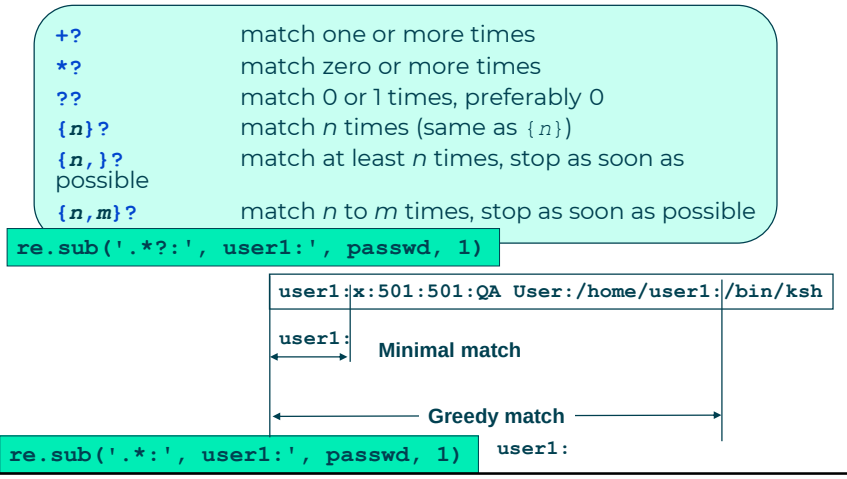
```
df = '/dev/sd3d  135398  69683  52176  57%  /home/stuff'
m = re.search('''
        ^(?P<fs>\S+)   \s+ (?:\d+)       \s+ (?:\d+) \s+
         (?P<avail>\d+)\s+ (?P<cap>\d+)% \s+ (?P<mnt>\S+)
             ''', df, re.X)
if m:
    gd = m.groupdict()
    print('{0[mnt]} ({0[fs]}) has {0[avail]} ({0[cap]}%) free'.
          format(gd))
```

```
/home/stuff (/dev/sd3d) has 52176 (57%) free
```

One of the most eagerly awaited features of Python was 'named captures'. The ability to give a name to capturing parentheses groups.
The captured values are placed into a dictionary with the keys set to the capture names and the values of whatever was matched.
They can be used as back-references in an re.sub replacement string using `(?P=name)`.
By the way, the trailing **`re.X`** option allows a new-line (or any white-space) to be embedded in the regular expression and ignored.

Minimal matches

- **A minimal match ends with a question mark**
- Match as few times as possible



A minimal quantifier with match with as few characters as possible, whereas a maximal (the default) will attempt to match with as many as possible.
The descriptions of these quantifiers may seem strange, for example you might think that a minimal match of zero times will always match nothing! The key point to remember is that a regular expression will match the whole pattern if it possibly can, so a minimal pattern will match different numbers of characters, depending on the rest of the pattern. For example, in the text: "123456AAAAAA":

\d*	matches 123456
\d*?	matches, but $& is empty
\d{2}\d*?A	matches 123456A

In the last example, the \d{2} is followed by a minimal match, and is followed by the letter 'A'. In this case, the minimal match has no choice but to match 4 decimal characters, if the whole pattern is to match. Given this text and pattern, the maximal match will produce the same result.
The example above, shows a typical problem with greedy matches where we want to replace the first field, but end up replacing all

except the final field. Incidentally, the traditional solution (in awk, for example) would be:

```
re.sub('[^:]*:','eric:',passwd,1)
```

Multi-line matches

- **Python allows searching a pattern across multiple lines**
  - Just search a text which contains embedded `"\n"` characters
- **However, Python is normally cautious:**
  - `.` character class does not match newline
  - `.*` will only go until the end of the line
- **`re.S` flag treats `"\n"` as a normal character**
  - `.` matches any character, including newline
  - `.*` will match until the end of the search text
  - `\s` and `\S` shortcuts are not affected

Normally . does not match a new-line character. With the /s option, a . does match a new-line, but not at the end of text. Remember that character class shortcut, \s always matches [\t\n\r\f] and \S always matches [^ \t\n\r\f], regardless of the /s option. This also applies to the POSIX character class [:space:], but not [:blank:]. Character class [:ascii:] also includes the new-line character, whatever its position.

Multi-line matches

- **Normally, ^ and $ mean: start and end of search text**
- **For multiple-line matches, this may be inconvenient**
- You may want to search the start and end of each line in the text
- **`re.M` matches ^ and $ for each line within search text**
- Can be combined with `re.S` option

```
Fred
DPM
Clive
Tom
Dick
Harry
```

```
all = open('names.txt').read()

m = re.search(r'(^dpm|^james)', all,re.I)
if m:
    print('File starts with', m.groups())

m = re.search(r'(^dpm|^james)', all, re.I|re.M)
if m:
    print('A line in the file starts with', m.groups())
```

```
A line in the file starts with ('DPM',)
```

Normally, python will treat the entire search text as a one string and not check for newlines within the string. If you read a file line by line, this does not matter. If you read an entire file into one variable containing multiple lines (slurping), it does matter, and you can use the m flag to handle this special case. Be careful though, \m means that . matches a line boundary, if a "\n" appears at the end-of-line there will be two matches, one for the "\n" and one for end-of-text ($).

Alternatives to ^ and $

\A matches beginning of string

- Will not match multiple times with /m

\Z matches end of string, or before a "\n"

- The "\n" is optional

\z matches end of string

- The real end-of-string, the "\n" is significant

> "Knowing how long a piece of string is
> can only be useful if you can find its ends"

These special characters are not Python specific, and are used by many other RE engines.

Comments in regular expressions

- **Delimit comments with `(?#`*comment*`)`**

```
re.search(
  r'\d{3}(?# 3 digits)\s(?# space)\d{3,5}(?# 3-5 digits)',
  txt)
```

- **The `re.X` flag allows comments in REs**
- White-space in the regular expression is discarded
- Comments until end-of-line allowed

```
re.search(
    '''
      \d{3}       # 3 digits
      \s          # space
      \d{3,5}     # 3-5 digits
    ''',
    txt, re.X)
```

Whereas most people will agree that comments are a GOOD THING in 'normal' code, not everyone agrees that comments inside regular expressions add clarity. They can actually confuse the code, as seen in the first example.
The use of the /x modifier makes things more obvious, and that in itself is good. However, this modifier discards white-space, so if you need to include white-space in your RE you have to 'escape' (\ ) it.

Lambda in re.sub

- **The re.sub method can take a function as a replacement**
  - Passes a match object to the function
  - The return value is the value substituted

```
import re

numbers = ['zero', 'wun', 'two', 'tree', 'fower',
           'fife', 'six', 'seven', 'ait', 'niner']
alphas  = ['alpha', 'bravo', 'charlie', 'delta', 'echo',
           'foxtrot', 'golf', 'hotel', 'india', 'juliet',
           'kilo', 'lima', 'mike', 'november', 'oscar', 'papa',
           'quebec', 'romeo', 'sierra', 'tango', 'uniform',
           'victor', 'whisky', 'xray', 'yankee', 'zulu']

codes = {str(i):name for i, name in enumerate(numbers)}
codes.update({name[0].upper():name for name in alphas})
reg = 'WG07 OKD'

result = re.sub(r'(\w)',
                lambda m: codes[m.groups()[0]] + ' ', reg)
```

The regular expression substitute operations (re.sub and re.subn) allow the second parameter to be a function (this is where the replacement string normally goes). The function can be a pre-defined named function, or a lambda.
In the example, we are constructing a dictionary. The keys come from a list of character ranges, and values are from the NATO phonetic alphabet. Care must be taken to ensure these lists are in the correct order.
The substitution matches any alphanumeric, and copies this to a match object by the use of a parentheses group `(\w)`. The single character is then used as a key to the dictionary.
The output in this case will be:

```
whisky golf zero seven  oscar kilo delta
```

Look-around assertions

- **Do not consume the pattern, or capture**
- Look-ahead: Positive: `(?=`*pattern*`)` Negative: `(?!`*pattern*`)`
- Look-behind: Positive: `(?<=`*pattern*`)` Negative: `(?<!`*pattern*`)`
- **Without look-arounds**

```
var = '<h1>This is a header</h1>'
m = re.search(r'<([hH]\d)>.*</\1>', var)
print("Matched: ", m.group())
```

```
Matched: <h1>This is a header</h1>
```

- **With look-arounds**

```
var = '<h1>This is a header</h1>'
m = re.search(r'(?<=<([hH]\d)>).*(?=</\1>) ', var)
print("Matched: ", m.group())
```

```
Matched: This is a header
```

Look-ahead and look-behind can be used as an extra safeguard in a regular expression: match this, but only if (not) ...
Perl supports the following zero-width look-around assertions,

(?=pattern)	positive look-ahead
(?!pattern)	negative look-ahead
(?<=pattern)	positive look-behind
(?<!pattern)	negative look-behind

In the first example, without look-arounds, we have a parentheses group which captures the header number part of the tag, and then uses a back-reference \1 to match the same header. The RE is enclosed in ! because we have a literal / as part of the pattern. This matches the HTML statement, including the tags.
The second example looks behind for the first tag and looks ahead for the second. The value of \1 is not affected by the extra parentheses groups. This time the tags are not included in the match.
Look-behinds can be a performance overhead in Python.
Unusually, there is a restriction with the Python regular expression engine - it (currently) does not support variable length look-behinds. This restriction is shared with Perl and Ruby, but Java and

PCRE do allow the patterns to be variable lengths. Other engines (and earlier versions) do not allow alternation in look-behinds. Look-aheads have no such restrictions.

Another example

Subtract 1 from the 2nd field for:

- `.log` files where the user (3rd field) is `root`
- `.dat` files where the type (4th field) is `system`

```
for line in open('log.txt'):
   nline = re.sub(
     '(?:(?<=\.log,)(\d+)(?=,root,)) |
      (?:(?<=\.dat,)(\d+)(?=.*,system$))',
     lambda m: str(int(m.group(1))-1) if m.group(1) \
               else str(int(m.group(2))-1), line, re.X)

   print(nline, end = "")
```

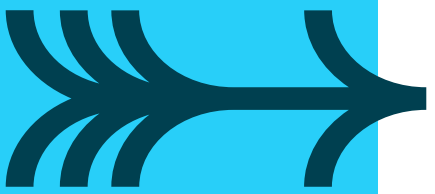
```
accounts.dat,2,user12,system
apache.log,1682,apache,daemon
var.log,23,root,user
profile.dat,57,home,user
payroll.dat,887,prd,system
```

→

```
accounts.dat,1,user12,system
apache.log,1682,apache,daemon
var.log,22,root,user
profile.dat,57,home,user
payroll.dat,886,prd,system
```

This RE uses an alternation | and each side must be grouped. We don't want to capture them, so `(?: ... )`.
The actual substitution is executed code and it uses the capture generated in the match.
Finally, the **`re.X`** modifier allows us to split the RE over two lines to fit it on the slide!

SUMMARY



- Modifiers alter the effect of the RE
- Side effect variables and capturing can be controlled
- Minimal matches use a ? after the quantifier
- Multi-line matching uses re.M and re.S
- Look-around assertions
- Substitution with interpolation
- Substitution with expressions