Beta Regex Unit

John M. Morrison

August 15, 2024

Contents

0 Introduction

A *regular expression* is a specification for a textual pattern. In the next couple of sections, we will develop the machinery necessary for you to add this powerful tool to your programming repertoire.

The first section will deal with the simplest regular expression, the `character class`; said item is a wildcard for a single character. All regular expressions are built out of character classes and regular expressions.

This chapter will be Python-specific, but most of what you see can be readily used in other languages, including Java and JavaScript.

A thorough and general treatment of regular expressions can be found in [**hedwig**]. A very useful resource is the website regex101. This site allows you to enter strings and regular expressions and see how they match. On this site, you can experiment with other dialects of regular expressions that appear in languages such as Java, JavaScript and Go. You will see that these dialects have quite a bit in common.

1 Character Wildcards

The first step in this process is to understand character classes, which represent a wildcard for a single character.

Here are examples of classes of characters we might like to represent.

- an alphabetical character
- a punctuation mark
- a decimal digit
- lower–case letters
- upper–case letters
- an octal digit
- a hex digit
- any old list of characters you'd like to create
- any whitespace character
- an asciicographical range of characters

We will now learn how to create character wildcards.

The simplest character class is just a character by itself. For example, the character `a` is the character class standing for the character `a`. This is called a *literal* character class. NB: Below we will discuss some character classes that are special characters (metacharacters), and how to deal with these.

The next simplest character class shows an explicit list of characters. Oddly enough, these are known as *list character classes* For example, `[aeiou]` matches any of the lower–case letters a, e, i, o or u. Notice how this character class lives in a "house" made of `[]`. The characters `]` and `[` are metacharacters. They play the role of being the exterior walls of a list character class's house. We will make a complete list of metacharacters for list character classes in later in this section.

Now let's see how this works in Python. We will import the `re`, or regular expression, module into our session.

```
>>> import re
>>> re.search("[aeiou]", "z")
>>> print(re.search("[aeiou]", "z"))
None
>>> print(re.search("[aeiou]", "a"))
<re.Match object; span=(0, 1), match='a'>
```

So, what happened here? In the first instance, we checked to see if our character class matched with the letter `'z'`. It did not, so the graveyard object `None` got returned. In the second instance it matched the character `'a'`. What got returned is a *match object*. It told us it found the letter `'a'` between indices 0 and 1 in the string `"a"`. This match object is truthy, so it can be used as predicate in a loop or a conditional statement because `None` is falsy.

We said that any literal character, save for magic characters, is a character class. Continuing the session above we see this.

```
>>> print(re.search("a", "z"))
None
>>> print(re.search("a", "a"))
<re.Match object; span=(0, 1), match='a'>
>>> print(re.search("a", "A"))
None
```

Note the case-sensitivity. Here is another magic character.

```
>>> print(re.search("^", "a"))
<re.Match object; span=(0, 0), match=''>
```

Whoa! We see a match, but it appears to be right at index 0. This is because the character `^` is a magic character: it means start-of-line. Now see this.

```
>>> print(re.search("$", "a"))
<re.Match object; span=(0, 0), match=''>
```

The dollar sign is a special character: it means end-of-line. Behold this fiasco.

```
>>> print(re.search("(", "a(bc"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/... /lib/python3.12/re/__init__.py",
  line 177, in search
```

```
    return _compile(pattern, flags).search(string)
           ^^^^^^^^^^^^^^^^^^^^^^^^

  File "/Library/Frameworks/... /lib/python3.12/re/__init__.py",
  line 307, in _compile
    p = _compiler.compile(pattern, flags)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "/Library/Frameworks/... /lib/python3.12/re/_compiler.py",
  line 743, in compile
    p = _parser.parse(p, flags)
        ^^^^^^^^^^^^^^^^^^^^^^^

  File "/Library/Frameworks/... /lib/python3.12/re/_parser.py",
  line 972, in parse
    p = _parse_sub(source, state, flags & SRE_FLAG_VERBOSE, 0)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "/Library/Frameworks/... /lib/python3.12/re/_parser.py",
  line 453, in _parse_sub
    itemsappend(_parse(source, state, verbose, nested + 1,
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "/Library/Frameworks/... /lib/python3.12/re/_parser.py",
  line 857, in _parse
    raise source.error("missing ), unterminated subpattern",
re.error: missing ), unterminated subpattern at position 0
```

Note that we omitted part of the paths to the errors for the sake of typographical sanity. You can replicate this yourself to see the gory details. This fierce ululation occurs because parentheses are magic characters. Any special character can be defanged by preceding it with a backslash (aka ”whack”).

```
>>> print(re.search("\(", "a(bc"))
<stdin>:1: SyntaxWarning: invalid escape sequence '\('
<re.Match object; span=(1, 2), match='('>
>>>
>>> print(re.search(r"\(", "a(bc"))
<re.Match object; span=(1, 2), match='('>
```

Note that when using a \ in a character class, you should use a raw string. You can see that the opening paren was found between indices 1 and 2.

Refresher: asciicographical Recall that every character has a byte (ASCII/UTF-8) value that is an integer. If you are unsure about the ASCII value of a character, use Python’s ord function.

```
>>> ord("a")
97
>>> ord("z")
```

```
122
>>> ord("A")
65
```

The asciicographical order on characters orders characters by their ASCII value.

Ranges Another basic type of character class is the *range* character class. Ranges are asciicographical ranges. Ranges are always shown inside of a list character class. The - character indicates a range. Inside of a list character class, use to put a - inside of a list character class.

```
>>> print(re.search("[a-f]", "piffle"))
<re.Match object; span=(2, 3), match='f'>
```

You can put several ranges inside of a list character class.

```
>>> re.search("[a-gn-z]", "hiccup")
<re.Match object; span=(2, 3), match='c'>
```

The - sign has some friends; we show them here. These are all special characters when used inside of a list character class.

`[`	begin a character class formed with a list of characters
`]`	end a character class formed with a list of characters
`(`	begin a group (more about this shortly). Use `\(`.
`)`	end a group (more about this shortly). Use `\(`.
`-`	produces a range (see below)
`\`	defangs any magic character (ex: `\[` for a `[` character)
`^`	special “not” character

Let’s get sNOTty! Putting a `^` as the first character in a list character class indicates anything BUT. For example `[^aeiou]` will match any character except the five lower-case regular vowels. Note that the `^` becomes an ordinary character unless used as the first character inside of a list character class.

There are some special character classes you will want to know about. Here we show a table of them. Use a raw string if the regex string contains a `\`.

`.`	any character except newline
`\d`	decimal digit
`\D`	not a decimal digit
`\s`	whitespace
`\S`	anything but whitespace
`\w`	any alphanumeric character or an underscore
`\W`	anything but an alphanumeric character or underscore

2 Regexes with Juxtaposition

A *regular expression* (regex) is a specification of a textual pattern. Regular expressions are built from regular expressions and character classes. Character classes are the "atoms" and regular expressons are "molecules." Certain characters have special meaning in the language of regular expression, so we show them here.

Basic Metacharacters (One Keystroke)	
Metacharacter	Action
`[`	begin delimiting a list character class
`]`	end delimiting a list character class
`{`	begining of a specific-number quantifier
`}`	end of a specific-number quantifier
`^`	beginning-of-line boundary
`$`	end-of-line boundary
`|`	or operator
`\`	escape character The escape character can make other characters into metacharacters, or it can remove magic from a metacharacter.
`(`	left delimiter for a group
`)`	right delimiter for a group
`.`	any single character except for a newline
`* + ?`	quantifiers

If you want ot use these as a character class, precede them with a
.

The most basic way to build a regular expression is by using *juxtapositon*, which simply means, "putting next to." Its meaning in a regex is "and then immediately" Let's show a simple example.

```
>>> import re
>>> index = re.search(r"[a-f][g-k]", "geese in a gaggle")
>>> index
<re.Match object; span=(12, 14), match='ag'>
```

Our regex specifies, "a letter a-f and then immediately a letter g-k." This is found between indices 12 and 14 in the form of `"ag"`. You can create a regex juxtaposing several character classes.

Boundaries This table shows boundaries. They can be used like character classes, but they are not characters *per se*. You will also hear them called *anchors*.

`^`	start of line
`$`	end of line
`\b`	word boundary
`\B`	not at the boundary of a word

Let's do some examples. You should create a Python session, follow along, and do insane experiments. Here we look for a word boundary then a z.

```
>>> s = "zealous buzzing zebras"
>>> re.search(r"\bz", s)
<re.Match object; span=(0, 1), match='z'>
```

Observe that the beginning of a line counts as a word boundary. Now let's look for z that is not next to a word boundary.

```
>>> re.search(r"\Bz", s)
<re.Match object; span=(10, 11), match='z'>
>>> s[10:]
'zzing zebras'
```

You can see it found the first z interior to a word.

Programming Exercises Test your solutions by having them act on a variety of string. Try hard to break your solutions.

1. Write a regex that recognizes `"foo"` as a stand-alone word.
2. Write a regex that will match words ending in `"try"`.
3. Write a regex that will match the words twit, tart, and toot.
4. The game of Battleship features coördinates formed from a letter A-I and a number 1-9. Write a regex that matches and coördiane word such as c8. It should not match an illegal set of coördintates such as K9. Can you make it case-insensitive?

3 Search Methods in `re`

So far, we have used `re.search` to obtian a match object. There are several other methods that are useful for learning about matches and for searching a string.

The method `re.findall` returns a list will all matches in it. We demonstrate it here.

```
>>> import re
>>> s = "abracadabra"
>>> regex = re.compile("a[a-g]")
>>> regex.findall(s)
['ab', 'ac', 'ad', 'ab']
```

It lives up to its advertised name.

Programming Exercise Write a program that takes these inputs.

- a regex
- a filename

Use `re.compile` because you will be looping. This will test your knowledge of file handling and apply what you just learned here. This program will go through the file line-by-line and do these things.

- If the line has a match for the regex, it will print out that line number.
- In addition, it will print out all of the matches it found on the line.

The method `re.match` must match from the very start of the string. Note this comparison vs. `re.search`.

```
>>> import re
>>> s = "barbarian"
>>> regex = re.compile("a[a-z]")
>>> regex.match(s)
>>> regex.search(s)
<re.Match object; span=(1, 3), match='ar'>
```

The method `re.fullmatch` matches from both ends. The entire string must match the regex or no match object is returned. We compare three methods here.

```
>> regex = re.compile("a[a-z]{3}")
>>> regex.match(s)
<re.Match object; span=(0, 4), match='abcd'>
>>> regex.search(s)
<re.Match object; span=(0, 4), match='abcd'>
>>> regex.fullmatch(s)
```

Notice no match on `re.fullmatch`.

4 Quantifiers

Suppose we want to match a Social Security number. We know what they look like: `XXX-XX-XXXX`, where each `X` is a decimal digit. We could do this: `\d\d\d-\d\d-\d\d\d\d`. It might even be a good idea to recognize it as a word by using `\b\d\d\d-\d\d-\d\d\d\d\b`.

```
>>> s = "I wonder if anyone has social security number 000-00-0000."
>>> import re
>>> match = re.search(r"\b\d\d\d-\d\d-\d\d\d\d\b", s)
>>> match
<re.Match object; span=(46, 57), match='000-00-0000'>
>>> s[46:57]
'000-00-0000'
```

What if we need to have 100 of some character class? Do we want to type the thing out 100 times? There must be a better way! This is where *quantifiers* play a role. Quantifiers are postfix unary operators, and they have precedence in the order of operations over juxtaposition. We will deploy one here.

```
>>> match = re.search(r"\b\d{3}-\d{2}-\d{4}\b", s)
>>> match
<re.Match object; span=(46, 57), match='000-00-0000'>
```

The quantifier {3} says, "match exactly 3 of my operand." Since
`d` represents a decimal digit,
`d`{3} will match three decimal digits. Note that a dash comes next. It is not a special character in the language of regular expressions; it only has magic powers inside of a list character class. You can easily see how the rest of it works.

```
>>> t = "This is muddy 321-44-32132."
>>> match = re.search(r"\b\d{3}-\d{2}-\d{4}\b", t)
```

Observe that we got no match here because of the fifth digit at the end, which does not match `\b`.

The quantifer `{n}` will match exactly `n` of its operand. It has a cousin `{m, n}` that will match at least `m` but not more than `n` of is operand. Here we show a table of quantifiers. All are postfix unary operators and all take precedence over the juxtapostion operation.

Quantifiers	
Operator	Action
`?`	expression appears 0 or 1 times
`+`	expression appears at one or more times consecutively
`*`	expression appears at zero or more times consecutively
`{n}`	expression appears exactly n times
`{m,n}`	expression appears at least m but not more than n times

4.1 Parentheses

Parentheses do two things in the world of regex. One useful function is that they can be used to override the order of operations. Consider the problem of writing a regex to match the string `"ab"` repeated one or more times. We want these to match `"ab"`, `"abab"` `"ababab"`, but not `"babab"` or `"baba"`. Let us build

1. word boundary
2. one or more of `"ab"`
3. word boundary

```
>>> import re
>>> try1 = r"\bab+\b"
>>> s = "ab"
>>> s = "There is an ab here"
>>> t = "We have an abab."
>>> u = "Yep, and an ababab, too"
>>> x = "The word babab is not a tree, it's a baobab."
>>> y = "Gimme some baba cake."
```

We hoe for a match on `s`, `t`, and `u`, but not on `x`, or `y`. Letterrip!

```
>>> try1 = r"\bab+\b"
>>> re.search(try1, s)
<re.Match object; span=(12, 14), match='ab'>
>>> re.search(try1, t)
>>> re.search(try1, u)
>>> re.search(try1, x)
>>> re.search(try1, y)
```

Ugh. Not what we wanted. We expected no match on `x` or `y`. This is because `ab+` matches an `a` followed by zero or more `b`s. Parentheses are needed to override. So let's try again.

```
>>> try2 = r"\b(ab)+\b"
>>> re.search(try2, s)
<re.Match object; span=(12, 14), match='ab'>
>>> re.search(try2, t)
<re.Match object; span=(11, 15), match='abab'>
>>> re.search(try2, u)
<re.Match object; span=(12, 18), match='ababab'>
>>> re.search(try2, x)
>>> re.search(try2, y)
```

We see what we expected. So, if we want a quantifier to act on a part of a regex, we must enclose that part in parentheses.

Searching Files For this section, you should download the file `scrabble.txt` from the site `Book Items`. This is a scrabble dictionary and we will do some searching in it. Note that all of the words are set in upper-case, one word to a line.

Compilation If you are using a regex in a loop, you should *compile* your regular expresson to a regular expression object using `re.compile`. This object is more efficient than using `re.search` on every turn of the loop.

Now suppose we are doing a crossword and we have the clue "Make your mark!" We know some letters; we have `P..CT.U...ON`. We create the program `cross.py` that will search `scrabble.txt` for candidates.

```
import re

def main():
python starry.py
19
Traceback (most recent call last):
  File "/Users/morrison/book/procedural_python/recut/pn7code/starry.py", line 13, in <module
    main()
  File "/Users/morrison/book/procedural_python/recut/pn7code/starry.py", line 10, in main
    print(total(nums))
          ^^^^^^^^^^^
  File "/Users/morrison/book/procedural_python/recut/pn7code/starry.py", line 4, in total
    out += k
TypeError: unsupported operand type(s) for +=: 'int' and 'list'regex = re.compile("P..CT....
with open("scrabble.txt", "r") as fp:
    for line in fp:
        if regex.search(line):
            print(line, end="")
```

Now run it.

```
unix> python cross.py
PUNCTUATION
PUNCTUATIONS
REPUNCTUATION
REPUNCTUATIONS
```

You can sharpen the search with a `\b` on either side, because you know that there are no letters before the P or after the N. You should modify this program and try it. Remember, we are in a crossword puzzle so we know how many letters the solution has.

Here ia silly and amusing question. Are there scrabble words in which the letters, a, b, c, d, and e appear in order? First we build a regex. We don't care what happens before a, so we won't use a ^ at the start. After a, we allow any number of characters to appear. For one character, use .; for zero or more we use the quantifier *. So we start with `a.*`. Now let's put in the rest to get `a.*b.*c.*d.*e`. We don't care what happens afterward, so no boundary on either side. Here is our code, `order.py`.

```
import re
def main():
    regex = re.compile("A.*B.*C.*D.*E")
    with open("scrabble.txt", "r") as fp:
        for line in fp:
            if regex.search(line):
                print(line, end="")

if __name__ == "__main__":
    main()
```

Wonder what line numbers they are on? Let's do it using `enumerate`.

```
import re
def main():
    regex = re.compile("A.*B.*C.*D.*E")
    with open("scrabble.txt", "r") as fp:
        for num, line in  enumerate(fp, 1):
```

```
unix> python order.py
ABSCONDED
ABSCONDER
ABSCONDERS
ABSTRACTEDNESS
AMBUSCADE
AMBUSCADED
AMBUSCADER
AMBUSCADERS
AMBUSCADES
```

```
            if regex.search(line):
                print(num, line, end="")

if __name__ == "__main__":
    main()
```

Vroooommm!!

```
461 ABSCONDED
462 ABSCONDER
463 ABSCONDERS
568 ABSTRACTEDNESS
4544 AMBUSCADE
4545 AMBUSCADED
4546 AMBUSCADER
4547 AMBUSCADERS
4548 AMBUSCADES
```

note the use of `enumerate(fp, 1)` to start enumerating with line 1 in the file.

Programming Exercise Write a program that takes these inputs.

- a regex
- a filename

Use `re.compile` because you will be looping. This will test your knowledge of file handling and apply what you just learned here. This program will go through the file line-by-line and do these things.

- If the line has a match for the regex, it will print out that line number.
- In addition, it will print out all of the matches it found on the line.

4.2 Using or

The operator — means ”or”. When using it, ALWAYS enclose the things you are “orring” in parens! This is a strict style expectation; adhere to it. It protects you from all manner of stupidity. The or operator is piggy and if you do not use parens, you do not control its ardor.

Let’s plunge in with an example. Notice how we escape the magic character . to defang its magic (any character).

```
regex = re.compile(r"^(Morrison|Sheck) is a nut\.$")
>>> regex.search("Morrison is a nut")    #no period, no match
>>> regex.search("Morrison is a nut.")
<re.Match object; span=(0, 18), match='Morrison is a nut.'>
>>> regex.search("Sheck is a nut.")
<re.Match object; span=(0, 15), match='Sheck is a nut.'>
```

Programming Exercise Write a regex that will match phone numbers fo the form (XXX)XXX-XXXX or XXX-XXX-XXXX. Note that the first digit of the groups of three cannot be a 0 or a one. Example: (102)566-6778 is not a legit number, nor is 201-133-4558.

5 June 14

What’s June 14? It’s flag day. And there are some useful flags you can use to refine your regex searching. Each has a full name and a short name.

Flags		
Flag	Short	Action
`re.IGNORECASE`	`re.I`	Ignore case when matching.
`re.MULTLINE`	`re.M`	multiline mode
`re.DOTALL`	`re.S`	Make . match `\\n` too.
`re.NOFLAG`		Use for no flag. It’s handy for default value in a function.

They can be combined using the or operator `|`. For example, the expression `re.IGNORECASE|re.MULTILINE` causes case to be ignored and for things to be matched in multiline mode. Let’s now see how to use some of these.

6 Groups and `finditer`