Regular Expressions in Python

Margaret Gratian

NIH NATIONAL CANCER INSTITUTE

5/25/2023

Disclaimer

- NIH Base Project Numbers are used as examples in these slides. These numbers were randomly made for this tutorial and are not connected with any real funded projects.

Regular Expressions

- "A tiny, high specialized programming language embedded inside Python"[1] that can be used to match text patterns
- May also be called REs or regexes or regex patterns
- The power of of regexes is matching *patterns* in text
  - Example: find all strings that look like emails, e.g., strings that match the format text@text.text. Writing a regex pattern for this could help us find margaret.gratian@nih.gov and margaretgratian@gmail.com
- Available through the "re" module, which is already built into Python but must be explicitly imported ("import re")

[1] https://docs.python.org/3/howto/regex.html

Introductory Example – Matching Exact Text

Example 0

```
1  # Text we want to search in
2  my_text = 'hello world, how are you today?'
3
4  # The pattern we want to match
5  # Here we are matching an exact phrase
6  pattern = r'hello world'
7
8  # Searching for the pattern using the RE search method
9  match = re.search(pattern, my_text)
10
11 print(match)
```

```
<re.Match object; span=(0, 11), match='hello world'>
```

Note: this is not a good use case for a regex. Using '==' will do the same and is faster. This example is chosen to illustrate the parts of a regex before introducing the complexity of building a regex pattern.

Why Would I Want to Use a Regex?

- The Python string library provides a lot of functionality for working with text
- What value do regexes provide?
  - Match varying sets of characters
  - Match repetitions in text
  - Match different options or conditions
  - Match specific locations of text
  - Matching text patterns but only returning specific parts of the text

Key Concepts

- Patterns
- Pattern methods
- Match objects
- Match object methods

Key Concepts

- **Patterns**
  - A string (which is compiled into a pattern object*) that specifies the format of text you want to match
- Pattern methods
- Match objects
- Match object methods

*The details of how a string is converted to a pattern object or how to do this yourself via re.compile() are not necessary for most use cases. The common regex module functions you will use will automatically do this step for you.

Patterns and Matches and findall()

Example 1

```
1  # Text we want to search
2  my_text = "personal email: margaretgratian@gmail.com, work email: margaretgratian@nih.gov"
3  print(my_text)
```

```
personal email: margaretgratian@gmail.com, work email: margaretgratian@nih.gov
```

```
1  # Pattern - the format we expect an email to follow
2  pattern = r'\w+@\w+\.\w+'
```

*

```
1  # Searching for the pattern using the RE findall method
2  match = re.findall(pattern, my_text)
3
4  print(match)
```

```
['margaretgratian@gmail.com', 'margaretgratian@nih.gov']
```

*Note the r' before the pattern indicates we are creating a *raw* Python string. This helps us deal with issues that might come up with the backslash symbol. More on this later.

Building Patterns

- Building patterns to match text involves using special characters and formatting – remember, regexes are effectively their own language!
- How you format a regex will depend on whether you want to match a single character or full words and phrases, alphanumeric or special characters, repetitions of text, whitespaces, newlines, and so on…
- Following Python's tutorial, we'll start with matching single characters and learn about how to handle some special characters along the way before moving on to more elaborate patterns

Matching Characters

- Most characters match themselves exactly, such as alphanumeric characters and some special characters
- Regexes are sensitive to capitalization
- Special characters that can be readily matched include !, @, %, &
- You can quickly tell if you can't match a special character this way because the match will fail or you'll get an error

Example 2a - Matching Characters

```
1 # Match an lower case alpha character
2 pattern = "a"
3 match = re.findall(pattern, "this is a sentence")
4 print(match)
```

```
['a']
```

```
1 # Match an upper case alpha character
2 pattern = "T"
3 match = re.findall(pattern, "This is a sentence")
4 print(match)
```

```
['T']
```

```
1 # Match a number
2 pattern = "1"
3 match = re.findall(pattern, "This is a number: 1")
4 print(match)
```

```
['1']
```

```
1 # Match a !
2 pattern = "!"
3 match = re.findall(pattern, "Match this exclamation point!")
4 print(match)
```

```
['!']
```

Matching Special Characters

- In the regex language, some special characters `. ^ $ * + ? { } [ ] \ | ( )` are *metacharacters* and instead of matching themselves, are used to construct regexes that can handle different variations on text
  - In fact, use of these characters is a critical part of using regexes to their full potential! We'll come back to this…
- What happens if you try to match one of these special metacharacters the way you match other characters?

Matching Metacharacters – Error Example

```
 4 pattern = "?"
 5 match = re.findall(pattern, "Can a regex detect a quesiton mark?")
 6 print(match)
```

```
---------------------------------------------------------------------------
error                                     Traceback (most recent call last)
Input In [13], in <cell line: 5>()
      1 # Cannot match a question mark this way
      2 # Uncomment to see this causes an error
      4 pattern = "?"
----> 5 match = re.findall(pattern, "Can a regex detect a quesiton mark?")
      6 print(match)

File ~/miniconda3/envs/crs_analysis_env/lib/python3.9/re.py:241, in findall(pattern,
    233 def findall(pattern, string, flags=0):
    234     """Return a list of all non-overlapping matches in the string.
    235
    236     If one or more capturing groups are present in the pattern, return
   (...)
    239
    240     Empty matches are included in the result."""
--> 241     return _compile(pattern, flags).findall(string)
```

This part of the error message is giving us a hint about what **?** does

```
    671     raise source.error("multiple repeat",
    672                        source.tell() - here + len(this))

error: nothing to repeat at position 0
```

Correctly Matching Metacharacters

- The backslash "\" is one of the most powerful metacharacters, as it can be used to *escape* metacharacters so you can match them

Example 2b - Matching Special Metacharacters

```
1  # Matching a question mark with the backslash \
2  pattern = "\?"
3  match = re.findall(pattern, "Match this: ?")
4  print(match)
```

```
['?']
```

```
1  # Matching a question mark with the backslash \
2  pattern = "\$"
3  match = re.findall(pattern, "This costs $2.00 while this costs $3.15")
4  print(match)
```

```
['$', '$']
```

Matching the Backslash \ Special Metacharacter, "The Backslash Plague"

Example 2c - Matching the Backslash with Escapes

```
# Create a string with a backlash
# Note that even in a Python string, we need to escape it or
# we will get errors

backslash_str = "Match this:\\"

# Note this output is misleading and makes us think both \\ are there
backslash_str
```

```
'Match this:\\'
```

```
# Observe when we print the string, we see in fact that
# it only has one backslash
print(backslash_str)
```

```
Match this:\
```

```
# To create the match pattern, we need four backslashes!
match = re.findall("\\\\", backslash_str)

# This output will be a bit misleading...
print(match)
```

```
['\\']
```

```
# Let's access the specific element in the list
# and print to see what it really is
print(match[0])
```

```
\
```

https://docs.python.org/3/howto/regex.html#the-backslash-plague

Matching the Backslash \ Special Metacharacter, A Better Way

Example 2d - Matching the Backslash with Raw Strings

```
# Create a string with a backlash
# Note the r to indicate this is raw string notation

backslash_raw_str = r"Match this: \ "

print(backslash_raw_str)
```

```
Match this: \
```

```
# To create the match pattern, we again use a raw string
# We still need one escape to escape the metacharacter
match = re.findall(r"\\", backslash_raw_str)

print(match[0])
```

```
\
```

Note that in backslash_raw_str, we add a white character space at the end, otherwise we'd end up escaping the ending quotation mark! If the backslash did not appear at the end or before a special character, this would not be necessary.

Raw Strings

- **Recommendation** – always write your regex pattern as a raw string
- It is essential in many cases, especially when you use special metacharacters to build your pattern
- It's a good practice to maintain, even if your pattern does not require them

Raw String Example

```
1 raw_string = r"This is a raw string because of the r before the quotation marks."
2 print(raw_string)
```

```
This is a raw string because of the r before the quotation marks.
```

Matching Character Classes

- What if you don’t need to match specific text or a specific character, but instead want to match any alpha character, or any numeric character?
- The [ ] are used to build character classes, in which you specify a set of characters that you want to match
- Upper and lower case alpha characters can be matched with [a-zA-Z]
- Numeric characters can be matched with [0-9]

Matching Character Classes – Alpha Characters

Example 4a - Matching Character Classes - Alpha Characters

```
# Create a pattern to match a, b, or c in text
pattern = r"[abc]"
match = re.findall(pattern, "a dog, a rabbit, and a cat")
print(match)
```

```
['a', 'a', 'a', 'b', 'b', 'a', 'a', 'c', 'a']
```

```
# Create a pattern to match a, b, or c in text, this time using a range
pattern = r"[a-c]"
match = re.findall(pattern, "a dog, a rabbit, and a cat")
print(match)
```

```
['a', 'a', 'a', 'b', 'b', 'a', 'a', 'c', 'a']
```

Matching Character Classes – Upper and Lower Case

Example 4b - Matching Character Classes - Upper and Lower Cases

```
# Match lower case
pattern = r"[a-z]"
match = re.findall(pattern, "The quick brown fox jumped over the lazy dog")
print(match)
```

```
['h', 'e', 'q', 'u', 'i', 'c', 'k', 'b', 'r', 'o', 'w', 'n', 'f', 'o', 'x', 'j', 'u', 'm', 'p', 'e', 'd', 'o', 'v',
'e', 'r', 't', 'h', 'e', 'l', 'a', 'z', 'y', 'd', 'o', 'g']
```

```
# Did we catch the T?
print("T" in match)
```

```
False
```

```
# Match upper and lower case
pattern = r"[a-zA-Z]"
match = re.findall(pattern, "The quick brown fox jumped over the lazy dog")
print(match)
```

```
['T', 'h', 'e', 'q', 'u', 'i', 'c', 'k', 'b', 'r', 'o', 'w', 'n', 'f', 'o', 'x', 'j', 'u', 'm', 'p', 'e', 'd', 'o',
'v', 'e', 'r', 't', 'h', 'e', 'l', 'a', 'z', 'y', 'd', 'o', 'g']
```

```
# Did we catch the T?
print("T" in match)
```

```
True
```

Matching Character Classes – Numeric Characters

Example 4C - Matching Character Classes - Numeric

```
1  # Create a pattern to match a, b, or c in text
2  pattern = r"[0-9]"
3  match = re.findall(pattern, "1 dog, 2 rabbits, and 30 cats")
4  print(match)
```

```
['1', '2', '3', '0']
```

Matching Character Classes – Special Metacharacters

- Metacharacters are not active inside a character class (except for the extra-special backslash \), so they can be matched as you would alphanumeric characters

Example 4d - Matching Character Classes - Metacharacters

```
1 # Create a pattern to match a, b, or c in text
2 pattern = r"[$.?]"
3 match = re.findall(pattern, "Is the dog food $30.00?")
4 print(match)
```

```
['$', '.', '?']
```

Complementing the Character Class

- Take the complement via the ^ symbol at the beginning to match anything *not* listed in the class

Example 4e - Complementing the Character Class

```
1  # Create a pattern to match a, b, or c in text
2  pattern = r"[^0-9]"
3  match = re.findall(pattern, "1 dog")
4  print(match)
```

```
[' ', 'd', 'o', 'g']
```

Special Sequences with the Backslash \

`\d`
Matches any decimal digit; this is equivalent to the class `[0-9]`.

`\D`
Matches any non-digit character; this is equivalent to the class `[^0-9]`.

`\s`
Matches any whitespace character; this is equivalent to the class `[ \t\n\r\f\v]`.

`\S`
Matches any non-whitespace character; this is equivalent to the class `[^ \t\n\r\f\v]`.

`\w`
Matches any alphanumeric character; this is equivalent to the class `[a-zA-Z0-9_]`.

`\W`
Matches any non-alphanumeric character; this is equivalent to the class `[^a-zA-Z0-9_]`.

These sequences can be included inside a character class. For example, `[\s,.]` is a character class that will match any whitespace character, or `','` or `'.'`.

https://docs.python.org/3/howto/regex.html#matching-characters

Repetition

- The characters *****, **+**, and **?** can be used to match repetitions
  - * will match a character (or character class or sequence) *zero or more* times
  - + will match *one or more* times
  - ? Will match exactly *zero or one* times
- The format {m, n} can be used to specify a range of matches, i.e., at least m repetitions and at most n
- What is the value of this?

Example 5a - The Power of Specifying Repetitions

```
1 pattern = r"colou?r"
2 match = re.findall(pattern, "color and colour are two different, but correct spellings")
3 print(match)
```

```
['color', 'colour']
```


Slide typo correction added – pattern should be colou?r, not colou*r which would also match “colouuur”

Understanding Differences in *, +, and ?

Example 5b - Differences in the Repetition Characters

```
# Using * to grab 0 or more alphanumeric characters
# Note that it will grab whitespace and punctuation
pattern = r"\w*"
match = re.findall(pattern, "This is a sentence with letters and some numbers such as 3, 5, 10, and 19")
print(match)
```

```
['This', '', 'is', '', 'a', '', 'sentence', '', 'with', '', 'letters', '', 'and', '', 'some', '', 'numbers', '', 'suc
h', '', 'as', '', '3', '', '', '5', '', '', '10', '', '', 'and', '', '19', '']
```

```
# Now we'll use the + to specify one or more alphanumeric characters
pattern = r"\w+"
match = re.findall(pattern, "This is a sentence with letters and some numbers such as 3, 5, 10, and 19")
print(match)
```

```
['This', 'is', 'a', 'sentence', 'with', 'letters', 'and', 'some', 'numbers', 'such', 'as', '3', '5', '10', 'and', '1
9']
```

```
# Now we'll use the ? to specify exactly zero or one alphanumeric characters
pattern = r"\w?"
match = re.findall(pattern, "This is a sentence with letters and some numbers such as 3, 5, 10, and 19")
print(match)
```

```
['T', 'h', 'i', 's', '', 'i', 's', '', 'a', '', 's', 'e', 'n', 't', 'e', 'n', 'c', 'e', '', 'w', 'i', 't', 'h', '',
'l', 'e', 't', 't', 'e', 'r', 's', '', 'a', 'n', 'd', '', 's', 'o', 'm', 'e', '', 'n', 'u', 'm', 'b', 'e', 'r', 's',
'', 's', 'u', 'c', 'h', '', 'a', 's', '', '3', '', '', '5', '', '', '1', '0', '', '', 'a', 'n', 'd', '', '1', '9',
'']
```

Key Concepts

- Patterns
- **Pattern methods**
  - Methods available for searching text once you have a pattern
- Match objects
- Match object methods

Common Pattern Methods

Method/Attribute	Purpose
`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.
`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.

We'll also cover:

- fullmatch()

Tables from https://docs.python.org/3/howto/regex.html

findall()

- Returns a list of matching strings

Example 6 - findall()

```
# Building a pattern to search for any numbers in the text
pattern = r"\d+"
match = re.findall(pattern, "This is a sentence with letters and some numbers such as 3, 5, 10, and 19")
print(match)
```

```
['3', '5', '10', '19']
```

```
# Building a pattern to search for non-whitespace
pattern = r"\S+"
match = re.findall(pattern, "Hello! How are you? I hope you've been well!!")
print(match)
```

```
['Hello!', 'How', 'are', 'you?', 'I', 'hope', "you've", 'been', 'well!!']
```

match() and search()

- These two methods provide similar functionality, but match will search only at the beginning of a string whereas search() checks for the match anywhere in the string
  - There's also a fullmatch() method that checks that the entire string matches
- They return None if the string does not match the pattern
- They will also search for the first occurrence of this before returning
- Both of these methods return what's called a **match object**
  - We'll cover this concept in the next section

Match documentation: https://docs.python.org/3/library/re.html#re.Pattern.match
Search documentation: https://docs.python.org/3/library/re.html#re.Pattern.search
Match vs. search vs. full match: https://docs.python.org/3/library/re.html#search-vs-match

match()

Example 7 - match()

```
1  # Search for something that must appear at the beginning of a string
2  pattern = r"Dr."
3
4  match1 = re.match(pattern, "Dr. Bunsen Honeydew and his assistant Beaker")
5  print(match1)
6
7  match2 = re.match(pattern, "Beaker works with Dr. Honeydew")
8  print(match2)
```

```
<re.Match object; span=(0, 3), match='Dr.'>
None
```

```
1  # Search for something that must appear at the beginning of a string
2  pattern = r"R01"
3
4  match = re.match(pattern, "R01123456")
5  print(match)
```

```
<re.Match object; span=(0, 3), match='R01'>
```

search()

Example 8 - search()

```
# Search for something that can appear anywhere
pattern = r"Dr."

match1 = re.search(pattern, "Dr. Bunsen Honeydew and his assistant Beaker")
print(match1)

match2 = re.search(pattern, "Beaker works with Dr. Honeydew")
print(match2)
```

```
<re.Match object; span=(0, 3), match='Dr.'>
<re.Match object; span=(18, 21), match='Dr.'>
```

```
# Search for something that can appear anywhere
# Note that it will still only find the first R01...why?
pattern = r"R01"

match = re.search(pattern, "R01123456, R0198765")
print(match)
```

```
<re.Match object; span=(0, 3), match='R01'>
```

finditer()

- Searches for all matches and returns an *iterator** of match objects that can be looped over
- Use this if you want to find all matches vs. just the first occurrence

Example 9 - finditer()

```
# Search for something that can appear anywhere and get all times it appears
pattern = r"R01"

matches = re.finditer(pattern, "R01123456, R0198765")

# This shows that we'll get an interator object back
print(matches)
```

```
<callable_iterator object at 0x12283eb20>
```

```
# Loop over the iterator to get the matches
for match in matches:
    print(match)
```

```
<re.Match object; span=(0, 3), match='R01'>
<re.Match object; span=(11, 14), match='R01'>
```

*We haven't covered the concept of iterators in Python yet but the details here are not important. But, if you are curious, see: https://www.freecodecamp.org/news/how-and-why-you-should-use-python-generators-f6fb56650888/

fullmatch() – with case insensitivity

Example 10a - fullmatch()

```
# Require that the entire string must match
pattern = r"PI"

match1 = re.fullmatch(pattern, "PI",  re.IGNORECASE)
print(match1)

match2 = re.fullmatch(pattern, "hospital coordinator", re.IGNORECASE)
print(match2)
```

Note the inclusion of this parameter so that we can match regardless of case!

```

match4 = re.fullmatch(pattern, "pi", re.IGNORECASE)
print(match4)

match5 = re.fullmatch(pattern, "HOSPITAL COORDINATOR", re.IGNORECASE)
print(match5)
```

```
<re.Match object; span=(0, 2), match='PI'>
None
None
<re.Match object; span=(0, 2), match='pi'>
None
```

Note that if you didn't care about capitalization, you could also check for string equality via string1 == string2.

A Non-Regex Option for Matching Exact Text

Example 10b - String Equality

```
1  str1 = "pi"
2  str2 = "PI"
3
4  print(str1 == str2)
5
6  str3 = "pi"
7  print(str1 == str3)
```

```
False
True
```

fullmatch() vs. ==

Example 10c - When fullmatch() is better than String Equality

```
1  pattern = r"(PI\s*)+"
2
3  match1 = re.fullmatch(pattern, "pi",  re.IGNORECASE)
4  print(match1)
5
6  match2 = re.fullmatch(pattern, "Pi", re.IGNORECASE)
7  print(match2)
8
9  match3 = re.fullmatch(pattern, "Pi pi pi", re.IGNORECASE)
10 print(match3)
11
12 match4 = re.fullmatch(pattern, "PI PI PI PI PI PI", re.IGNORECASE)
13 print(match4)
```

```
<re.Match object; span=(0, 2), match='pi'>
<re.Match object; span=(0, 2), match='Pi'>
<re.Match object; span=(0, 8), match='Pi pi pi'>
<re.Match object; span=(0, 17), match='PI PI PI PI PI PI'>
```

To do the same with ==, you would need to check for every possible scenario!

More on Matching Words

Example 10d - Matching a string that might be in another string

Say we want to find the string PI, as we have in previous examples. However, we have cases where PI is not the only thing in a String; we also have cases where PI is the only word; and we also have cases where PI might exist within another word (but we don't want to match it).

Match a word at the start of a string: "PI Bunsen Honeydew will..." with "^pi\s+"

Match a word at the end of a string: "The (PI)" with "\s+pi$"

Match a word that is the entire string: "PI" with "^pi$"

Match a word within the string: "The PI is..." with "\s+pi\s+"

All of the above also ensure we do not match a word if its contained in another word: e.g., do not match hos(pi)tal.

```
pattern = r"\s+pi\s+|^pi\s+|^pi$|\s+pi$"

match1 = re.search(pattern, "PI Bunsen Honeydew will...",  re.IGNORECASE)
print(match1)

match2 = re.search(pattern, "The PI",  re.IGNORECASE)
print(match2)

match3 = re.search(pattern, "PI",  re.IGNORECASE)
print(match3)

match4 = re.search(pattern, "The PI is",  re.IGNORECASE)
print(match4)

match4 = re.search(pattern, "hospital",  re.IGNORECASE)
print(match4)
```

```
<re.Match object; span=(0, 3), match='PI '>
<re.Match object; span=(3, 6), match=' PI'>
<re.Match object; span=(0, 2), match='PI'>
<re.Match object; span=(3, 7), match=' PI '>
None
```

Key Concepts

- Patterns
- Pattern methods
- **Match objects**
  - A data structure containing information about where a match starts and ends and the string it matched
- Match object methods

Matches

- Has the format <re.Match object; span=(start, end), match = “string”>
- From here, there are an additional set of methods available to extract information from this object

```
<re.Match object; span=(0, 2), match='PI'>
```

Key Concepts

- Patterns
- Pattern methods
- Match objects
- **Match object methods**
  - Methods to get information about a match object

Common Match Methods

Method/Attribute	Purpose
`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

Tables from https://docs.python.org/3/howto/regex.html

Common Match Methods Example

Group: get the text that matched

Start: get the start position

End: get the end position

Span: get the start and end position

Example 11 - group(), start(), end(), span()

```
# Match a word (1 or more a-z characters, upper or lower case, only)
pattern = r"[a-zA-Z]+"

match = re.search(pattern, "1 dog")
print(match)
```

```
<re.Match object; span=(2, 5), match='dog'>
```

```
# Use group to get the text that matched
match.group()
```

```
'dog'
```

```
# Use start() to get the position of where the matched text begins
match.start()
```

```
2
```

```
# Use end() to get the position of where the matched text ends
match.end()
```

```
5
```

```
# Use span() to get the position of the start and end of the matched text
match.span()
```

```
(2, 5)
```

Other Important Concepts

- More metacharacters
- Groups
- Resources for building regexes
- When not to use a regex

Other Important Concepts

- **More metacharacters**
- Groups
- Resources for building regexes
- When not to use a regex

The or | Metacharacter

Example 12a - The | Metacharacters

```
1  # Look for an R01 equivalent DP1, DP2, DP5, R01, R37, R56, RF1, RL1, U01
2  pattern = r"DP1|DP2|DP5|R01|R37|R56|RF1|RL1|U01"
3
4  match1 = re.findall(pattern, "R01CA12345, DP1CA09876, R21CA34560")
5  print(match1)
6
7  match2 = re.findall(pattern, "R01CA12345")
8  print(match2)
9
10 match3 = re.findall(pattern, "DP1CA09876")
11 print(match3)
```

```
['R01', 'DP1']
['R01']
['DP1']
```

The ^ and $ Metacharacters

^ specifies that a pattern must exist at the start of a string

$ specifies that a pattern must exist at the end of a string

Example 12b - The ^ Metacharacters

```
# Match at the beginning of text only
pattern = r"^NCI"

match1 = re.findall(pattern, "NCI is a government agency at the NIH")
print(match1)

match2 = re.findall(pattern, "CRS is a center within the NCI")
print(match2)
```

```
['NCI']
[]
```

Example 12c - The $ Metacharacters

```
# Match at the end of text only
pattern = r"NCI$"

match1 = re.findall(pattern, "NCI is a government agency at the NIH")
print(match1)

match2 = re.findall(pattern, "CRS is a center within the NCI")
print(match2)
```

```
[]
['NCI']
```

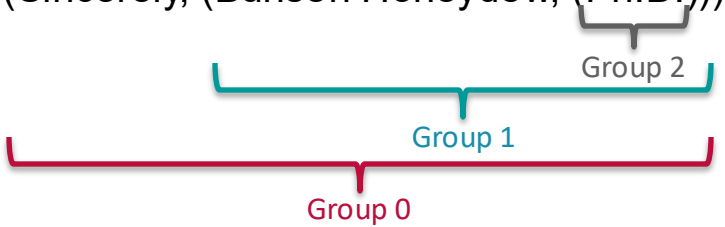
Other Important Concepts

- Using additional metacharacters
- **Groups**
- Resources for building regexes
- When not to use a regex

Groups using ( )

- Sometimes you don't just want to match some text, you also want to "dissect" the text into specific components
- For example, maybe you have text associated with a grant and the text "Abstract: " before each abstract. You might be interested in finding all the text that begins with "Abstract:" but just keeping the text that follows. You can do this with regexes and groups!
- The **( )** are what define a group within a pattern
- You can have as many groups as you want, and even have nested groups, e.g., say you want to capture someone's signature block, and then within that their title:

r"(Sincerely, (Bunsen Honeydew, (Ph.D.)))"



Multiple Groups Example

Example 12b - Multiple Groups

```python
# A pattern that captures a structure that might match a name
pattern = r"(\w+)\s(\w+)"

# Searching for the pattern using the RE findall method
match = re.match(pattern, "Firstname Lastname", re.IGNORECASE)
print(match)

# group(0) gives us the whole match
print(f"The entire match: {match.group(0)}")

# group(1) gives us the first group that matched
print(f"Group 1: {match.group(1)}")

# group(2) gives us the second group that matched
print(f"Group 2: {match.group(2)}")
```

```
<re.Match object; span=(0, 18), match='Firstname Lastname'>
The entire match: Firstname Lastname
Group 1: Firstname
Group 2: Lastname
```

Non-capturing Groups using (?...)

Example 12c - Non-capturing Groups

```
1  # This pattern uses a non-capturing group, where we look for a match in the text
2  # but we don't actually care about getting back this group
3  # This pattern looks for NCI project numbers, which are those that start with CA and are followed by some numbers
4  # We've made CA a non-capturing group because we don't care about that piece of information, outside of
5  # needing it to be present for us to grab the project number
6  pattern = r"(?:CA)(\d+)"
7
8  # Searching for the pattern using the RE findall method
9  #match = re.findall(pattern, "CA09876")
10
11 match = re.findall(pattern, "OD123456, CA109876, HL345678")
12 print(match)
```

```
['109876']
```

Named Groups

- Instead of referring to groups using numbers (which can get complicated to keep track of with overly nested groups), you can also give names to your groups
- Group names are assigned via **?P<group_name>**

Example 12d - Named Groups

```
# In this pattern, we are matching two things
# First, alpha characters, and exactly 2 of them
# Second, a base project number which is one or more digits
# We also assign names to these groups with the syntax ?p<group_name>
pattern = r"(?P<institute>[a-zA-Z]{2,2})(?P<base_project_num>\d+)"

# Search and assign groups to the text
match = re.search(pattern, "CA109876")
print(match)
```

```
<re.Match object; span=(0, 8), match='CA109876'>
```

```
# Access the groups
print(match.group('institute'))
print(match.group('base_project_num'))
```

```
CA
109876
```

Named Groups with finditer()

Example 12e - Mathing all Named Groups

```
# In this pattern, we are matching exactly 2 alpha characters
# and a base project number which is one or more digits
# Again, we assign them the names institute and base_project_ num
pattern = r"(?P<institute>[a-zA-Z]{2,2})(?P<base_project_num>\d+)"

# Return an iterator of match objects via the finditer() method
matches = re.finditer(pattern, "OD123456, CA109876, HL345678")
print(matches)
```

```
<callable_iterator object at 0x10a600520>
```

```
# Convert matches to a list
matches_list = list(matches)
```

```
# Loop over the list, accessing the institute and base_project_num
for match in list(matches_list):
    print(match.group('institute'))
    print(match.group('base_project_num'))
    print("\n")
```

```
OD
123456


CA
109876


HL
345678
```

Other Important Concepts

- Using additional metacharacters
- Groups
- Tips on building regexes
- When not to use a regex

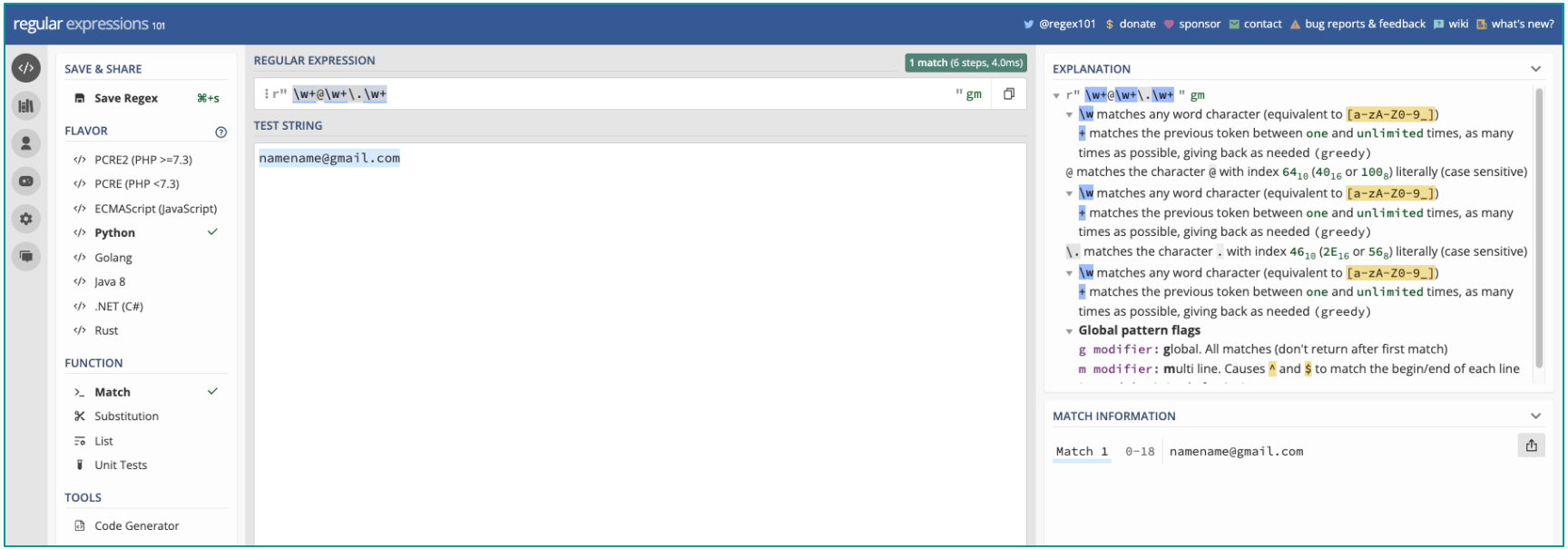
Other Important Concepts

- Using additional metacharacters
- Groups
- **Tips on building regexes**
- When not to use a regex

Building a Regex

- Start simple!
  - Think about the “base case” pattern you watch to match and build from there
  - Add in repetitions and other metacharacter functionality one at a time
- Think about the impact of whitespaces, new lines, and other text formatting elements on your match – regexes are sensitive to them!
- Use multiple regex patterns if you have several complex scenarios you need to capture (no need for one regex pattern to do everything)
- Consult online resources often to know all the special symbols available:
  - https://docs.python.org/3/library/re.html#regular-expression-syntax
- Once you think you have your regex built, test examples that you don’t want to match to ensure the regex isn’t capturing more than it should

Resources for Testing Regexes – regex101.com



Important: Be very cautious with what you type in, especially if you are building a regex for sensitive data. This is a **public website** and there is **no guarantee of security or protection** of what you type. If trying to match something in sensitive data and you want to test it here, think about how you can make your test case more generic or reduced to the simplest/smallest pattern you are trying to match.

Other Important Concepts

- Using additional metacharacters
- Groups
- Tips on building regexes
- **When not to use a regex**

When Not to Use Regexes

- If you can accomplish the same task with Python's built in functionality or the string library, this may be the better option[1] as it is usually much faster and much more readable. Examples include:
  - Matching a fixes string or single character
  - Replacing a single word or character
  - Deleting the occurrence of a character
- NLP libraries such as spaCy are better suited to complicated text processing tasks such as tokenization, finding adverbs, etc. (even though you can accomplish these with regexes)

[1]For more see: https://docs.python.org/3/howto/regex.html#use-string-methods

Questions?

NATIONAL CANCER INSTITUTE

www.cancer.gov

www.cancer.gov/espanol