

BINF 5007

Lecture 10: Regular Expression

Learning Objectives

1. Basics of Regular Expressions

- Understand regex syntax and applications in data analysis.

2. Using Regex in Python

- Apply ``re.search()`` and ``re.findall()`` for pattern matching.

3. Greedy vs. Non-Greedy Matching

- Differentiate and apply greedy and non-greedy techniques.

4. Parsing with Regex

- Solve real-world parsing tasks using regex.

5. Anchored Patterns

- Use anchors (``^``, ``$``) for precise pattern matching.

6. Compiled Patterns

- Improve efficiency with ``re.compile()`` for repetitive tasks.

Regular Expressions

In computing, a regular expression, also referred to as “regex” or “regexp”, provides a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor.

http://en.wikipedia.org/wiki/Regular_expression

← → ↻ en.wikipedia.org/wiki/Regular_expression

regular| 1/146

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

Regular expression

From Wikipedia, the free encyclopedia

"Regex" redirects here. For the comic book, see [Re:Gex](#).

A **regular expression**, **regex** or **regexp**^[1] (sometimes called a **rational expression**)^{[2][3]} is a sequence of **characters** that define a *search pattern*. Usually such patterns are used by **string searching algorithms** for "find" or "find and replace" operations on **strings**, or for input validation. It is a technique developed in **theoretical computer science** and **formal language theory**.

The concept arose in the 1950s when the American mathematician **Stephen Cole Kleene** formalized the description of a **regular language**. The concept came into common use with **Unix** text-processing utilities. Different **syntaxes** for writing **regular expressions** have existed since the 1980s, one being the **POSIX** standard and another, widely used, being the **Perl** syntax.

Regular expressions are used in **search engines**, search and replace dialogs of **word processors** and **text editors**, in text processing utilities such as **sed** and **AWK** and in **lexical analysis**. Many **programming languages** provide regex capabilities either built-in or via **libraries**.

I watch three climb before it's my turn. It's a tough one. The guy before me tries twice. He falls twice. After the last one, he comes down. He's finished for the day. It's my turn. My buddy says "good luck!" to me. I noticed a bit of a problem. There's an outcrop on this one. It's about halfway up the wall. It's not a

The match results of the pattern

```
(?<=\. ) {2,} (?=[A-Z ])
```

They're like Really smart "Find" or "Search"

https://en.wikipedia.org/wiki/Regular_expression

The Purpose of Regular Expressions

- Regular expressions can be used to match patterns within strings
 - Finding pieces of text within a larger document
- Regular expressions are composed of
 - **characters**
 - **character classes**
 - **metacharacters**
 - **quantifiers**
 - **assertions**

The Importance of Patterns in Biology

- **Biological Sequences as Strings:**

- DNA, RNA, and protein sequences are strings.
- Patterns we look for include:
 - Protein domains
 - DNA transcription factor binding motifs
 - Restriction enzyme cut sites
 - Degenerate PCR primer sites
 - Runs of mononucleotides

- **Beyond Sequence Data:**

- Patterns can also appear in:
 - Read mapping locations
 - Geographical sample coordinates
 - Taxonomic names
 - Gene names and accession numbers
 - BLAST searches

The Purpose of Regular Expressions

- In general the letters and characters match with themselves
- “Python” is going to match with “Python” (but not with “python”)
- The exceptions to this rule are **metacharacters**
- **Metacharacters** are characters that have a special meaning in the context to the regular expression:

. ^ \$ * + ? { } [] \ | ()

Regular Expression Quick Guide

anchors and boundaries

Anchor	Legend	Example	Sample Match
<code>^</code>	Start of string or start of line depending on multiline mode. (But when <code>[^</code> inside brackets, it means "not")	<code>^abc.*</code>	abc (line start)
<code>\$</code>	End of string or end of line depending on multiline mode. Many engine-dependent subtleties.	<code>.*? the end\$</code>	this is the end
<code>\A</code>	Beginning of string (all major engines except JS)	<code>\Aabc[\d\D]*</code>	abc (string... ...start)
<code>\z</code>	Very end of the string Not available in Python and JS	<code>the end\z</code>	this is...\n...the end
<code>\Z</code>	End of string or (except Python) before final line break Not available in JS	<code>the end\Z</code>	this is...\n...the end\n
<code>\G</code>	Beginning of String or End of Previous Match .NET, Java, PCRE (C, PHP, R...), Perl, Ruby		
<code>\b</code>	Word boundary Most engines: position where one side only is an ASCII letter, digit or underscore	<code>Bob.*\bcat\b</code>	Bob ate the cat
<code>\b</code>	Word boundary .NET, Java, Python 3, Ruby: position where one side only is a Unicode letter, digit or underscore	<code>Bob.*\b\кошка\b</code>	Bob ate the кошка
<code>\B</code>	Not a word boundary	<code>c.*\Bcat\B.*</code>	copycats

Characters

Character	Legend	Example	Sample Match
<code>\d</code>	Most engines: one digit from 0 to 9	<code>file_\d\d</code>	file_25
<code>\d</code>	.NET, Python 3: one Unicode digit in any script	<code>file_\d\d</code>	file_9ᳵ
<code>\w</code>	Most engines: "word character": ASCII letter, digit or underscore	<code>\w-\w\w\w</code>	A-b_1
<code>\w</code>	.Python 3: "word character": Unicode letter, ideogram, digit, or underscore	<code>\w-\w\w\w</code>	字-ま_𐄂
<code>\w</code>	.NET: "word character": Unicode letter, ideogram, digit, or connector	<code>\w-\w\w\w</code>	字-ま_𐄂
<code>\s</code>	Most engines: "whitespace character": space, tab, newline, carriage return, vertical tab	<code>a\s\b\sc</code>	a b c
<code>\s</code>	.NET, Python 3, JavaScript: "whitespace character": any Unicode separator	<code>a\s\b\sc</code>	a b c
<code>\D</code>	One character that is not a digit as defined by your engine's <code>\d</code>	<code>\D\D\D</code>	ABC
<code>\W</code>	One character that is not a word character as defined by your engine's <code>\w</code>	<code>\W\W\W\W\W</code>	*-+=)
<code>\S</code>	One character that is not a whitespace character as defined by your engine's <code>\s</code>	<code>\S\S\S\S</code>	Yoyo

<https://www.regexg.com/regex-quickstart.php>

Regular Expression Quick Guide

Quantifiers

Quantifier	Legend	Example	Sample Match
+	One or more	Version \w-\w+	Version A-b1_1
{3}	Exactly three times	\D{3}	ABC
{2,4}	Two to four times	\d{2,4}	156
{3,}	Three or more times	\w{3,}	regex_tutorial
*	Zero or more times	A*B*C*	AAACC
?	Once or none	plurals?	plural

Quantifier	Legend	Example	Sample Match
+	The + (one or more) is "greedy"	\d+	12345
?	Makes quantifiers "lazy"	\d+?	1 in 12345
*	The * (zero or more) is "greedy"	A*	AAA
?	Makes quantifiers "lazy"	A*?	empty in AAA
{2,4}	Two to four times, "greedy"	\w{2,4}	abcd
?	Makes quantifiers "lazy"	\w{2,4}?	ab in abcd

More Characters

Character	Legend	Example	Sample Match
.	Any character except line break	a.c	abc
.	Any character except line break	.*	whatever, man.
\.	A period (special character: needs to be escaped by a \)	a\.c	a.c
\	Escapes a special character	\\. * \\? \\\$ \\^ \\	. * ? \$ ^
\\	Escapes a special character	\\[\\{ \\} \\	[{ }

Logic

Logic	Legend	Example	Sample Match
	Alternation / OR operand	22 33	33
(...)	Capturing group	A(nt pple)	Apple (captures "pple")
\\1	Contents of Group 1	r(\\w)g\\1x	regex
\\2	Contents of Group 2	(\\d\\d)\\+(\\d\\d)=\\2\\+\\1	12+65=65+12
(?: ...)	Non-capturing group	A(?:nt pple)	Apple

Character Classes

Character	Legend	Example	Sample Match
[...]	One of the characters in the brackets	[AEIOU]	One uppercase vowel
[...]	One of the characters in the brackets	T[ao]p	Tap or Top
-	Range indicator	[a-z]	One lowercase letter
[x-y]	One of the characters in the range from x to y	[A-Z]+	GREAT
[...]	One of the characters in the brackets	[AB1-5w-z]	One of either: A,B,1,2,3,4,5,w,x,y,z
[x-y]	One of the characters in the range from x to y	[-~]+	Characters in the printable section of the ASCII table .
[^x]	One character that is not x	[^a-z]{3}	A1!
[^x-y]	One of the characters not in the range from x to y	[^ -~]+	Characters that are not in the printable section of the ASCII table .
[\\d\\D]	One character that is a digit or \\d\\D)+ a non-digit	[\\d\\D]+	Any characters, including new lines, which the regular dot doesn't match
[\\x41]	Matches the character at hexadecimal position 41 in the ASCII table, i.e. A	[\\x41-\\x45]{3}	ABE

Regular Expression Quick Guide

More White-Space

Character	Legend	Example	Sample Match
\t	Tab	T\tw{2}	T ab
\r	Carriage return character	see below	
\n	Line feed character	see below	
\r\n	Line separator on Windows	AB\r\nCD	AB CD
\N	Perl, PCRE (C, PHP, R...): one character that is not a line break	\N+	ABC
\h	Perl, PCRE (C, PHP, R...), Java: one horizontal whitespace character: tab or Unicode space separator		
\H	One character that is not a horizontal whitespace		
\v	.NET, JavaScript, Python, Ruby: vertical tab		
\V	Perl, PCRE (C, PHP, R...), Java: one vertical whitespace character: line feed, carriage return, vertical tab, form feed, paragraph or line separator		
\V	Perl, PCRE (C, PHP, R...), Java: any character that is not a vertical whitespace		
\R	Perl, PCRE (C, PHP, R...), Java: one line break (carriage return + line feed pair, and all the characters matched by \v)		

anchors and Boundaries

Anchor	Legend	Example	Sample Match
^	Start of string or start of line depending on multiline mode. (But when [*inside brackets], it means "not")	^abc.*	abc (line start)
\$	End of string or end of line depending on multiline mode. Many engine-dependent subtleties.	.*? the end\$	this is the end
\A	Beginning of string (all major engines except JS)	\Aabc[d\D]*	abc (string... ...start)
\z	Very end of the string Not available in Python and JS	the end\z	this is...\n...the end
\Z	End of string or (except Python) before final line break Not available in JS	the end\Z	this is...\n...the end\n
\G	Beginning of String or End of Previous Match .NET, Java, PCRE (C, PHP, R...), Perl, Ruby		
\b	Word boundary Most engines: position where one side only is an ASCII letter, digit or underscore	Bob.*\bcat\b	Bob ate the cat
\b	Word boundary .NET, Java, Python 3, Ruby: position where one side only is a Unicode letter, digit or underscore	Bob.*\bкошка\b	Bob ate the кошка
\B	Not a word boundary	c.*\Bcat\B.*	copycats

The Steps I Take Writing Regular Expressions

- Three Steps I take when writing a regular expression:
 - First, describe the pattern in English, e.g.
 - *"ACG followed by 0 or more characters followed by GCA"*
 - Second, what part of match do you want to extract, if any?
 - Third, translate into Python (or whatever language you're using)

The Regular Expression Module

- Before you can use regular expressions in your program, you must import the library using `import re`
- You can use `re.search()` to see if a string matches a regular expression
 - similar to using the `find()` method for strings
- You can use `re.findall()` to extract portions of a string that match your regular expression, similar to a combination of `find()` and slicing:
`var[5:10]`

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
 Return-Path: <postmaster@collab.sakaiproject.org>
 Received: from murder (mail.umich.edu [141.211.14.90])
 by frankenstein.mail.umich.edu (Cyrus v2.3.8) with
 LMTPA; Sat, 05 Jan 2008 09:14:16 -0500
 X-Sieve: CMU Sieve 2.3
 Received: from murder ([unix socket])
 by mail.umich.edu (Cyrus v2.2.12) with
 LMTPA; Sat, 05 Jan 2008 09:14:16 -0500
 Received: from holes.mr.itd.umich.edu (holes.mr.itd.umich.edu [141.211.14.79])
 by flawless.mail.umich.edu () with ESMTP id m05EEFR1013674;
 Sat, 5 Jan 2008 09:14:15 -0500
 ..
 ..
 Received: from prod.collab.uhi.ac.uk ([194.35.219.182])
 by paploo.uhi.ac.uk (JAMES SMTP Server 2.1.3) with SMTP ID
 899 for <source@collab.sakaiproject.org>;
 Sat, 5 Jan 2008 14:09:50 +0000 (GMT)
 Received: from nakamura.uits.iupui.edu (nakamura.uits.iupui.edu
 [134.68.220.122]) by shmi.uhi.ac.uk (Postfix) with ESMTP id A215243002
 for <source@collab.sakaiproject.org>; Sat, 5 Jan 2008 14:13:33 +0000 (GMT)
 Received: from nakamura.uits.iupui.edu (localhost [127.0.0.1])
 by nakamura.uits.iupui.edu (8.12.11.20060308/8.12.11) with ESMTP id
 m05ECJVp010329 for <source@collab.sakaiproject.org>; Sat, 5 Jan 2008 09:12:19 -
 0500
 Received: (from apache@localhost)
 by nakamura.uits.iupui.edu (8.12.11.20060308/8.12.11/Submit) id
 m05ECIaH010327 for source@collab.sakaiproject.org; Sat, 5 Jan 2008 09:12:18 -
 0500
 Date: Sat, 5 Jan 2008 09:12:18 -0500
 X-Authentication-Warning: nakamura.uits.iupui.edu: apache set sender to stephen.marquard@uct.ac.za using -
 f To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
<http://www.py4inf.com/code/mbox-short.txt>

We want to find:

Using `re.search()` Like `find()`

```
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if line.find('From:') >= 0:
        print(line)
```

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
..
..
```

```
import re
```

```
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line) :
        print(line)
```

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
..
..
```

<http://www.py4inf.com/code/mbox-short.txt>

Using `re.search()` Like `startswith()`

```
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if line.startswith('From:') :
        print(line)
```

```
From: stephen.marquard@uct.ac.za
..
..
```

```
import re

hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line) :
        print(line)
```

```
From: stephen.marquard@uct.ac.za
..
..
```

We fine-tune what is matched by adding **special characters** to regexp

Wild-Card Characters

- The **dot** character matches any character
- If you add the **asterisk** character, the character is “any number of times”

```
X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-DSPAM-Confidence: 0.8475
X-Content-Type-Message-Body: text/plain
```

Match the start of the
line

Many
times

Match any character

Fine-Tuning Your Match

Depending on how “clean” your data is and the purpose of your application, you may want to narrow your match down a bit

```
X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-Plane is behind schedule: two weeks
X-: Very short
```

Match the start of
the line

Many
times

Match any character

Output not in order of 'mbox-short.txt' , Color denotes a match

Fine-Tuning Your Match

Depending on how “clean” your data is and the purpose of your application, you may want to narrow your match down a bit

```
X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-Plane is behind schedule: two weeks
X-: Very short
```

Match the start of
the line

One or more
times

[^]X-\S+

Match any non-whitespace character

Matching and Extracting Data

- `re.search()` returns a `True/False` depending on whether the string matches the regular expression
- Which is why we can use it in the `if` or `elif`
- **What if we want to extract the match?**
- If we want the matching strings to be extracted, we use `re.findall()`

`[0-9]+`



One or more digits

```
>>> import re
>>> x = 'My 2 favorite numbers are 19 and 42'
>>> y = re.findall('[0-9]+', x)
>>> print(y)
['2', '19', '42']
```

Matching and Extracting Data

When we use `re.findall()`, it returns a list of zero or more substrings that match the regular expression

```
>>> import re
>>> x = 'My 2 favorite numbers are 19 and 42'
>>> y = re.findall('[0-9]+', x)
>>> print(y)
['2', '19', '42']
>>> y = re.findall('[AEIOU]+', x)
>>> print(y)
[] # empty list
```

Matching and Extracting Data

When we use `re.findall()`, it returns a list of zero or more substrings that match the regular expression – **Case Insensitive**

```
>>> import re
>>> x = 'My 2 favorite numbers are 19 and 42'
>>> y = re.findall('[0-9]+', x)
>>> print(y)
['2', '19', '42']
>>> y = re.findall('[AEIOU]+', x, re.IGNORECASE)
>>> print(y)
['a', 'o', 'i', 'e', 'u', 'e', 'a', 'e', 'a']
```

Warning: Greedy Matching

The **repeat** characters (***** and **+**) are greedy and try to match the largest possible string

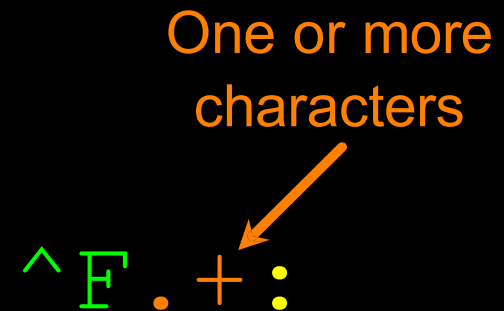
```
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^F.+:', x)
>>> print(y)
['From: Using the :']
```

Why not 'From:' ?

First character in
the match is an F

Last character in the
match is a :

One or more
characters

The diagram shows the regex pattern `^F.+:` with colored annotations. The `^` is green, `F` is green, `.` is orange, `+` is orange, and `:` is orange. An orange arrow points from the text "One or more characters" to the `+` operator. A green arrow points from the text "First character in the match is an F" to the `F`. A yellow arrow points from the text "Last character in the match is a :" to the `:`.

Non-Greedy Matching

Not all regular expression repeat codes are greedy!
If you add a ? character, the + and * do not act greedy then

```
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^F.+?:', x)
>>> print(y)
['From:']
```

One or more characters but not greedy

^ F . + ? :

First character in the match is an F

Last character in the match is a :

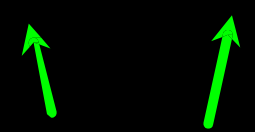
Fine-Tuning String Extraction

You can refine the match for `re.findall()` and separately determine which portion of the match is to be extracted by using parentheses

```
>>> x = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
```

```
>>> y = re.findall('\S+@\S+', x)
>>> print(y)
['stephen.marquard@uct.ac.za']
```

`\S+@\S+`



At least one
non-whitespace
character

Fine-Tuning String Extraction

Parentheses are not part of the match - but they tell where to **start** and **stop** what string to extract

```
>>> x = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
```

```
>>> y = re.findall('\S+@\S+', x)
```

```
>>> print(y)
```

```
['stephen.marquard@uct.ac.za']
```

```
>>> y = re.findall('^From (\S+@\S+)', x)
```

```
>>> print(y)
```

```
['stephen.marquard@uct.ac.za']
```

`^From (\S+@\S+)`



What's the difference in what we did
with the two regular expressions?

String Parsing Examples...

Remember

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

21 ↓ 31 ↓

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ', atpos)
>>> print(sppos)
31
>>> host = data[atpos+1 : sppos]
>>> print(host)
uct.ac.za
```

Extracting a host name
- using find and string
slicing

That's a lot of work!

Remember: The Double Split Pattern

Sometimes we split a line one way, and then grab one of the pieces of the line and split that piece again

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
words = line.split()
email = words[1]
pieces = email.split('@')
print(pieces[1])
```

```
stephen.marquard@uct.ac.za
['stephen.marquard', 'uct.ac.za']
'uct.ac.za'
```

That's a lot of work!

The Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
```

```
line = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
```

```
y = re.findall('@(\S+)', line)
```

```
print(y)
```

'@(\S+)'

```
['uct.ac.za']
```

Look through the string until you find an at sign

The Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
```

```
line = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
```

```
y = re.findall('@(\S+)', line)
```

```
print(y)
```

' @ (\ S +) '

['uct.ac.za']

Match non white space

Match 1 more more

The Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
```

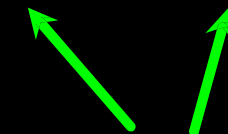
```
line = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
```

```
y = re.findall('@(\S+)', line)
```

```
print(y)
```

```
['uct.ac.za']
```

'@(\S+)'



Extract the non-blank white space

An Anchored Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
```

```
line = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
```

```
y = re.findall('^From .*@(\S+)', line)
```

```
print(y)
```

'^From .*@(\S+)'

```
['uct.ac.za']
```

Starting at the beginning of the line, look for the string 'From '

An Anchored Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
```

```
line = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
```

```
y = re.findall('^From .*@(\S+)', line)
```

```
print(y)
```

'^From . * @ (\ S +) '

```
['uct.ac.za']
```



Skip a bunch of characters, looking for an at sign

An Anchored Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
line = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
y = re.findall('^From .*@(\S+)', line)
print(y)
```

['uct.ac.za']

'^From .*@(\S+)'



Start extracting

An Anchored Regex Version

From stephen.marquard@uct.ac.za **Sat Jan 5 09:14:16 2008**

```
import re
line = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
y = re.findall('^From .*@(\S+)', line)
print(y)
```

'^From .*@(\S+).'

['uct.ac.za']

Match non-blank character

Match 1 or more

An Anchored Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
```

```
line = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
```

```
y = re.findall('^From .*@(\S+)', line)
```

```
print(y)
```

'^From .*@([^\s]+)'

```
['uct.ac.za']
```



Stop extracting

Escape Character

If you want a special regular expression character to just behave **normally** (most of the time) you prefix it with `\`

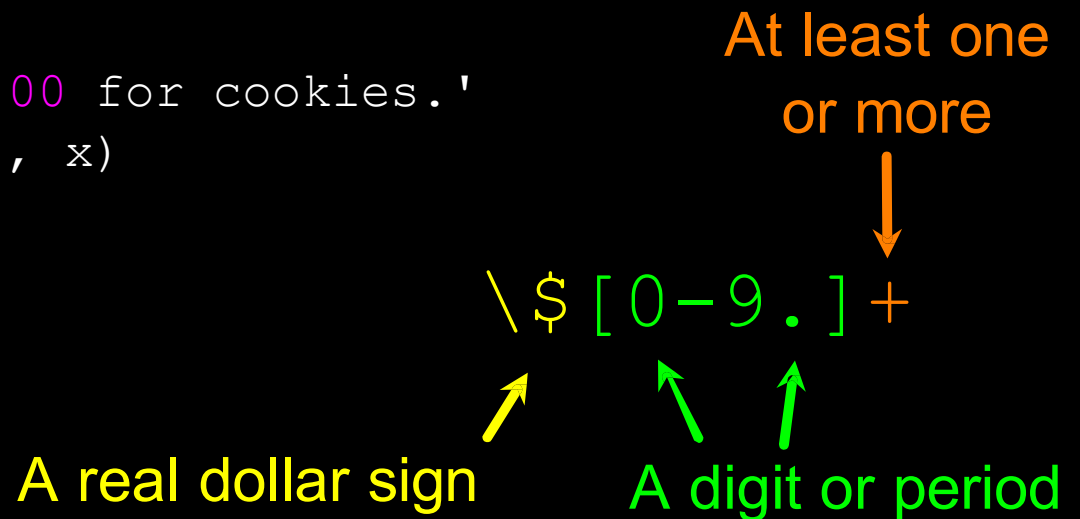
```
>>> import re
>>> x = 'We just received $10.00 for cookies.'
>>> y = re.findall('\$[0-9.]+', x)
>>> print(y)
['$10.00']
```

At least one
or more

\ \$ [0 - 9 .] +

A real dollar sign

A digit or period



Compiling a Pattern

- A pattern can be compiled (converted to an internal representation) to speed up the search
- This step is not mandatory but recommended for large amounts of text
- Let's see `findall` with a regular pattern and then with a “compiled” pattern

```
>>> re.findall("[Hh]ello","Hello world, hello Python,!")
['Hello', 'hello']
>>> rgx = re.compile("[Hh]ello")
>>> rgx.findall("Hello world, hello Python,!")
['Hello', 'hello']
```

- Compiled patterns have all methods available in the `re` module, `search`, `match`, `findall`

Your turn

Restriction enzymes are proteins that cut DNA at specific sequences. For example, the restriction enzyme EcoRI recognizes the sequence GAATTC and cuts between the G and the first A.

Write a Python program that identifies all occurrences of the EcoRI recognition site (GAATTC) in a given DNA sequence and prints the positions where the enzyme would cut the DNA.

Instructions:

Import the re module

Define the DNA sequence: Use a string to represent the DNA sequence.

Define the recognition site: Use a regular expression to represent the EcoRI recognition site.

Find all matches: Use the re.finditer() function to find all occurrences of the recognition site in the DNA sequence.

Print the positions: For each match, print the start and end positions where the enzyme would cut the DNA. The start() and end() methods could be useful.