Lecture 4
Regular Expressions/ Classes

Lecturer: Pieter De Bleser

Bioinformatics Core Facility, IRC

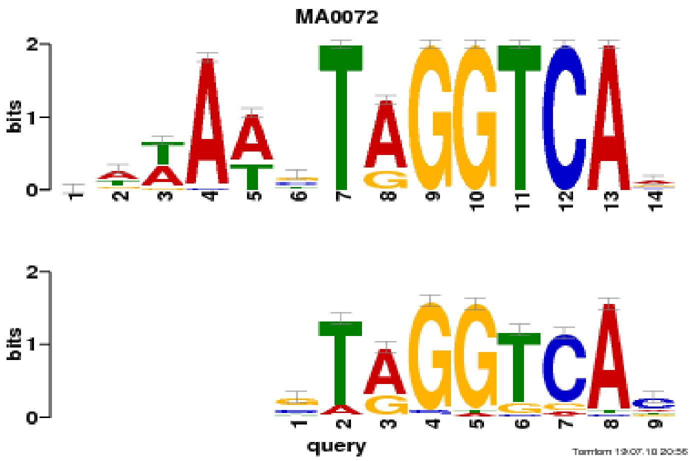
Slides derived from: I. Holmes, Department of Statistics, University of Oxford; M. Schroeder, M. Adasme, A. Henschel, Biotechnology Center, TU Dresden; S. Spielman, CCBB, University of Texas; **Jehan-François Pâris, jfparis@uh.edu**

Regular Expressions

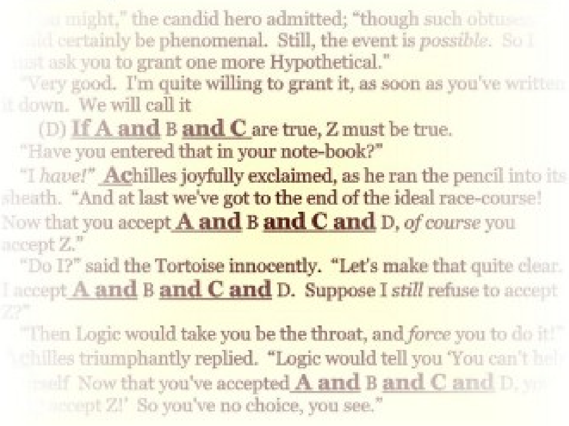
Introduction to regular expressions

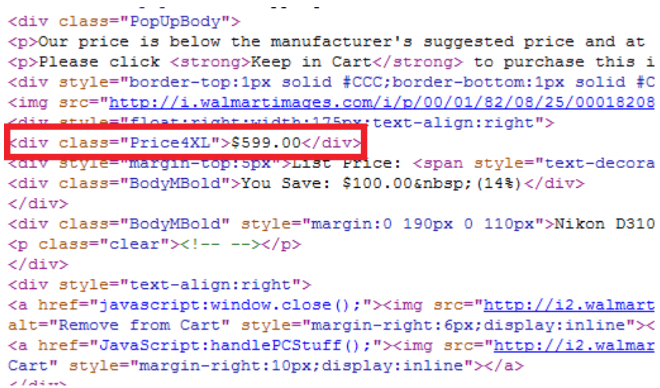
- In bioinformatics we often work with strings
- Regex: highly specialized “language” for matching strings
- In python: the re module
- Perl-style regular expressions
- Useful for scripts, shell scripts and command line
- In unix: “grep -P” uses similar commands
- In R: grepl, grep: set perl=TRUE

Why?



/([Ii]f|and)* [AC]+.(and)?/





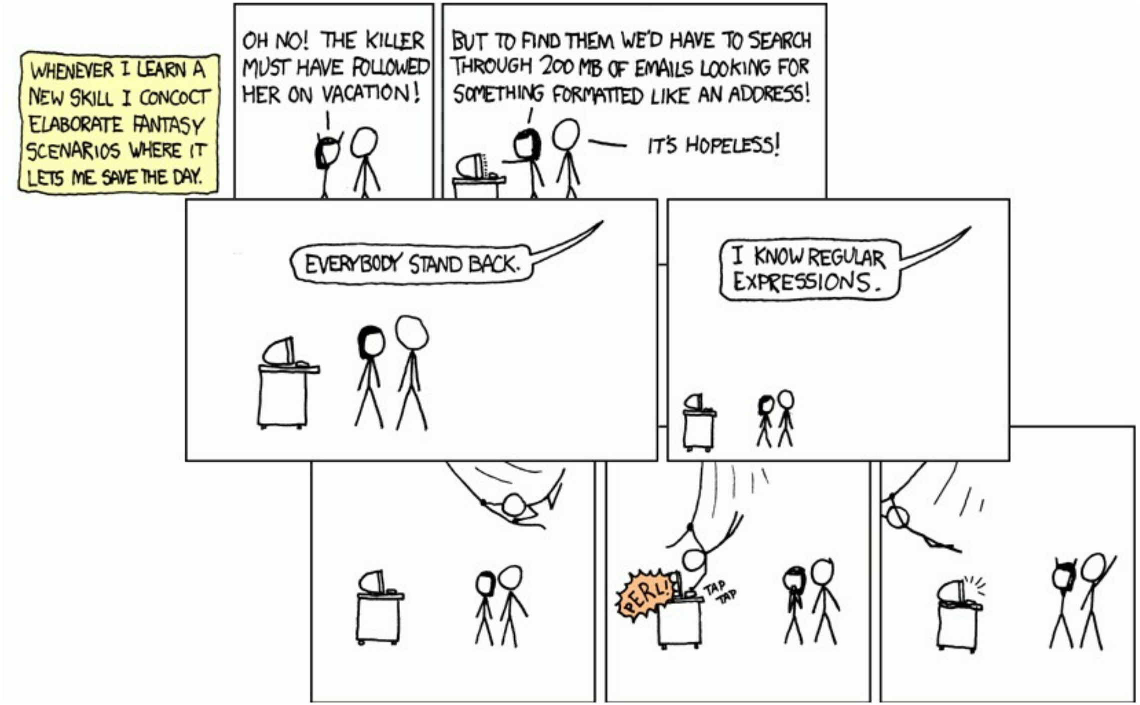
Simple example

- The simple way: use `re.search()`
- `re.search(`*`pattern, string, flags=0`*`)`
- Scan through *string* looking for a location where *pattern* produces a match, and return a corresponding **'Match object'** instance.
- Returns None if no position in the string matches the pattern

```
>>> import re
>>> re.search("hello", "oh, hello") != None
True
>>> re.search("hello", "oh, hwello") != None
False
>>> re.search("hello", "oh, Hello") != None
False
```

Outline

- Simple matching: basic rules
  - Some python issues
- Compiling expressions
- Additional methods
- The returned object
  - Getting more than one match
- Advanced stuff
  - Grouping options
  - Modifying strings



https://xkcd.com/208/

Metacharacters

- Character class: a set of characters we want to match
- Specified by writing in []
- Examples:
  - [abc], the same as [a-c]

```
>>> re.match('[abcd]',"d") != None
True
>>> re.match('[a-d]',"d") != None
True
```

- Use completing set:
  - [^5] – match everything but 5

```
>>> re.match('[^abcd]',"d") != None
False
>>> re.match('[^abcd]',"e") != None
True
```

The backslash \

- Regex:
  - Specifies defined classes
- In standard python:
  - Specifies escape characters (agreed patterns e.g. “\n”)
  - “\n”, “\t”, …
- “Problems”
  - When we want to match metacharacters
  - Clashes between Python and re definitions
  - Let’s ignore these problems for now

Python built-in use: escape characters

Escape Sequence	Meaning
****	**Backslash (\)**
\'	**Single quote (')**
\"	**Double quote (")**
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Form feed (FF)
\n	**ASCII Linefeed (LF)**
\N{name}	Character named *name* in the Unicode database (Unicode only)
\r	**ASCII Carriage Return (CR)**
\t	**ASCII Horizontal Tab (TAB)**
\uxxxx	Character with 16-bit hex value *xxxx* (Unicode only)
\Uxxxxxxxx	Character with 32-bit hex value *xxxxxxxx* (Unicode only)
\v	ASCII Vertical Tab (VT)
\ooo	Character with octal value *ooo*
\xhh	Character with hex value *hh*

These are standard strings identifiers, used by standard C

Backslash example:

```
>>> x = ""aabb""
  File "<stdin>", line 1
    x = ""aabb""
             ^
SyntaxError: invalid syntax
>>> x = "aabb"
>>> x
'aabb'
>>> x = "\"aabb\""
>>> x
'"aabb"'
```

Regex use: predefined classes

String	Class	Equivalent
\d	Decimal digit	[0-9]
\D	Non-digit	[^0-9]
\s	Any whitespace	[\t\n\r\f\v]
\S	Non-whitespace	[^ \t\n\r\f\v]
\w	Any alphanumeric	[a-zA-Z0-9_]
\W	Non alphanumeric	[^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 '.'.

Question

- Does a given DNA string contain a TATA-box-like pattern?
  - Define a TATA-box-like pattern as “TATAA” followed by 3 nucleotides and ends with “TT”

```
>>> import re
>>> def hasTataLike(string):
...     if (re.search("TATAA[ACGT][ACGT][ACGT]TT", string)):
...         return True
...     return False
...
>>> s = "ACGACGTTTACACGGATATAAGGGTTACGCGCTGTATAATGTGATCAGCTGATTCGAA"
>>> print (hasTataLike(s))
True
>>> s = "ACGACGTTTACACGGAAATAAGGGTTACGCGCTGTATAATGTGATCAGCTGATTCGAA"
>>> print (hasTataLike(s))
False
```

Matching any character

- The metacharacter “.” matches any character but newline.

```
>>> re.search("...", "ab\t") != None
True
>>> re.search("...", "ab\n") != None
False
>>> # Match two digits then any character then two non digits
>>> re.search("\d\d.\D\D", "98\tAD") != None
True
```

Repeats

- Character quantifiers:
  - “*” : the regex before can be matched zero or more times
    - ca*t matches ct, cat, caat, caaaaaaat …
    - Matching is “greedy”: python searches for the largest match
  - “+” : one or more times
    - ca+t matches cat but not ct
  - “?” once or zero times
- Specifying a range:
  - {m,n}: at least m, at most n
    - “a/{1,3}b” will match a/b, a//b, and a///b. It won’t match ab.
    - Omitting m is interpreted as a lower limit of 0, while omitting n results in an upper bound of infinity

Examples

- **.** – any single character except a newline
  - **bet.y** would match “betty”, “betsy” but not “bety”
  - **3\.1415** if the dot is what you’re after

- ***** - match the preceding character 0 or more times.
  - **fred\t*barney** matches fred, then any number of tabs, then barney. “fredbarney” is also matched
  - **.*** matches any character (other than newline) any number of times

Examples

- **+** is another **quantifier**, same as *, but the preceding items has to be matched >0 times
  - **fred +barney** - arbitrary number of spaces between fred and barney, but at least one
- **?** is another quantifer, this time meaning that zero or one matches are needed
  - **bamm-?bamm** will match “bammbamm“ and “bamm-bamm”, but only those two
  - Useful for optional prefix or suffix

Question

- Does a given DNA string contain a TATA-box-like pattern?
  - Define a TATA-box-like pattern as “TATAA” followed by 3 nucleotides and ends with “TT”

```
>>> import re
>>> def hasTataLike(string):
...     if (re.search("TATAA[ACGT]{3}TT", string)):
...         return True
...     return False
...
>>> s = "ACGACGTTTACACGGATATAAGGGTTACGCGCTGTATAATGTGATCAGCTGATTCGAA"
>>> print (hasTataLike(s))
True
>>> s = "ACGACGTTTACACGGAAATAAGGGTTACGCGCTGTATAATGTGATCAGCTGATTCGAA"
>>> print (hasTataLike(s))
False
```

Grouping patterns

- ( ) are a grouping meta-character
  - **fred+** matches fredddddd
  - **(fred)+** matches fredfredfredfred
  - **(fred)***
    - matches “Hello world” (or anything)

Question

At least two TATA like patterns?

```
>>> def multipleTataLike(string):
...     if (re.search("(TATAA[ACGT]{3}TT).*(TATAA[ACGT]{3}TT)",string)):
...         return True
...     return False
...
>>>
>>> s = "GATATAAGGGTTACGCGCTATAAGGGTTTTTTGTATAATGTGATCAGCTGATTCGAA"
>>> print (multipleTataLike(s))
True
>>> s = "ACGACGTTTACACGGAAATAAGGGTTACGCGCTGTATAATGTGATCAGCTGATTCGAA"
>>> print (multipleTataLike(s))
False
```

Alternatives

- **|** allows to separate between options
  - **fred|barney|betty** means that the matched string must contain fred **or** barney **or** betty
- **fred(|\t)+barney** matches fred and barney separated by one or more space/ tab
  - **fred(+|\t+)barney (?)**
    - Similar, but either all spaces or all tabs
  - **fred (and|or) barney (?)**
    - Matches “fred and barney” and “fred or barney”

Anchors

- \A
  - Match from the beginning of the string

```
>>> re.search('\AProsper', 'Prosper and live well') != None
True
>>> re.search('\AProsper', 'Party and live well') != None
False
>>> re.search('\APros', 'Pro') != None
False
>>> re.search('\APros', 'Prospe') != None
True
>>> re.search('\APros.+', 'Prosper and') != None
True
```

Anchors

- \Z
  - Match from the end of the string

```
>>> re.search('}\Z', '{block}') != None
True
>>> re.search('}\Z', '{block} ') != None
False
>>> re.search('}\Z', '{block}\n') != None
False
```

More anchors

- ^
  - Match the beginning of lines
- $
  - Match the end of lines
- Don’t forget to set the MULTILINE flag (more on flags later)

findall() matches *all* occurrences of a pattern, not just the first one as search() does

```
>>> gene_scores = "AT5G42600\t12.254\nAT1G08200\t302.1\n"
>>> print (gene_scores)
AT5G42600 12.254
AT1G08200 302.1

>>> re.findall("(\d+)$",gene_scores,re.MULTILINE)
['254', '1']
>>> re.findall("(\d)$",gene_scores)
['1']
```

Matching metacharacters

- Say we want to match the regex: (...${2,5}...)

```
>>> re.search("(...${2,5}...)","(ACG$$$$GCT)")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/pieterdb/anaconda3/lib/python3.7/re.py", line 183, in search
    ...
  File "/home/pieterdb/anaconda3/lib/python3.7/sre_parse.py", line 651, in
_parse
    source.tell() - here + len(this))
re.error: nothing to repeat at position 5
```

Use “\” before each metacharacter

Matching metacharacters

- Say we want to match the regex: (...${2,5}...)

```
>>> re.search("\(...\${2,5}...\)","(ACG$$$$GCT)") != None
True
>>> re.search("(...\${2,5}...)","ACG$$$$GCT") != None
True
>>> re.search("\(...\${2,5}...\)","ACG$$$$GCT") != None
False
```

The backspace plague

- “\” has a special use also in Python (not the re module)

```
>>> x="\"
  File "<stdin>", line 1
    x="\"
        ^
SyntaxError: EOL while scanning string literal
>>> x = "\\"
>>> x
'\\'
>>> print (x)
\
```

The backspace plague

- Regular expressions use the backslash character ('\') to indicate special cases.
- This conflicts with Python’s usage of the same character for the same purpose in string literals.

```
>>> y = "\section"
>>> y
'\\section'
>>> print(y)
\section
```

The backspace plague

- Say we want to match “\section”

```
>>> re.search("\section","\section") != None
False
>>> re.search("\\section","\ ection") != None
True
>>> re.search("\\section","\section") != None
False
>>> re.search("\\\\section","\section") != None
True
```

- One has to write '\\\\' as the RE string, because the regular expression must be \\, and each backslash must be expressed as \\ inside a regular Python string literal.

Using raw strings

- In Python, strings that start with r are raw, and “\” is not treated as a special character

```
>>> l = "\n"
>>> l
'\n'
>>> print(l)


>>> l=r"\n"
>>> l
'\\n'
>>> print(l)
\n
```

Using raw strings

- In Python, strings that start with r are raw, and “\” is not treated as a special character

```
>>> re.search(r"\\\\section","\section") != None
False
>>> re.search(r"\section","\section") != None
False
>>> re.search(r"\\section","\section") != None
True
```

When you really need “\” in your strings work with raw strings!

Compiling expressions

- Create an object that represents a specific regex and use it later on strings.

```
>>> p = re.compile('[a-z]+')
>>> p
re.compile('[a-z]+')
>>> print(p.match(""))
None
>>> m = p.match('tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

Compile vs. Static use

- Same rules for matching.
- The static use: Python actually compiles the expression and uses the result
  - When are going to the same expression many times compile once.
  - Running time difference is usually minor.
- Safer and more readable code
  - A good habit: compile all expression once in the same place, and use them later.
  - Reusable code, reduce typos.

Additional 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*.

The match object

- Both re.search and p.search (or match) return a match object.
  - Always has a True boolean value
  - The method re.finditer returns an iterable datastructure of match objects.
  - Useful methods:

Method/Attribute	Purpose
group()	Return the (sub)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

Getting many matches

```
>>> p = re.compile('\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')
['12', '11', '10']
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable_iterator object at 0x7f0c754d5d68>
>>> for match in iterator:
...     print (match.span())
...
(0, 2)
(22, 24)
(29, 31)
```

Compilation flags

- Add flexibility in the regex definition (at compilation time)
- Using more than one flag: add or between them
- Ignore case:
- Using re

```
>>> re.search("he","Hello",re.IGNORECASE)
<re.Match object; span=(0, 2), match='He'>
>>> p = re.compile("he",re.IGNORECASE)
>>> p
```

- Using compilation

```
re.compile('he', re.IGNORECASE)
>>> p.search("Hello")
<re.Match object; span=(0, 2), match='He'>
```

Compilation flags

- Locale:
  - Used for non-English chars (not relevant for this course)
- Multline (re.MULTLINE)
  - When this flag is specified, ^ matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline.
  - Similarly, the $ metacharacter matches either at the end of the string and at the end of each line (immediately preceding each newline).
- DOTALL
  - Makes the '.' special character match any character at all, **including a newline**

Grouping: getting sub-expressions

- Groups indicated with '(', ')' also capture the starting and ending index of the text that they match.
- This can be retrieved by passing an argument to `group()`, `start()`, `end()`, and `span()`.
- Groups are numbered starting with 0. Group 0 is always present; it’s the whole RE.
- Subgroups are numbered from left to right, from 1 upward.
- Groups can be nested; to determine the number, just count the opening parenthesis characters, going from left to right.

Example 1

What will span(X) return here?

```
>>> print (re.search("(ab)*AAA(cd)*","ababAAAcd").span())
(0, 9)
>>> print (re.search("(ab)*AAA(cd)*","ababAAAcd").group(0))
ababAAAcd
>>> print (re.search("(ab)*AAA(cd)*","ababAAAcd").group(1))
ab
>>> print (re.search("(ab)*AAA(cd)*","ababAAAcd").group(2))
cd
>>> print (re.search("(ab)*AAA(cd)*","ababAAAcd").group(3))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: no such group
```

Example 2

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

- group() can be passed multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups.

```
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

Example 3

- The groups() method returns a tuple containing the strings for all the subgroups, from 1 up to however many there are.

```
>>> m.groups()
('abc', 'b')
>>> len (m.groups())
2
```

Example 4

- The groups() method returns a tuple containing the strings for all the subgroups, from 1 up to however many there are.

```
>>> re.match("A(B+)C","ABBBBBC").groups()
('BBBBB',)
>>> re.match("A(B+)C","ABBBBBC").span(1)
(1, 6)
```

Modifying strings

Method/ Attribute	Purpose
split()	Split the string into a list, splitting it wherever the RE matches
sub()	Find all substrings where the RE matches, and replace them with a different string

Split

- Parameter: maxsplit
  - When *maxsplit* is nonzero, at most *maxsplit* splits will be made.
- Use re.split or p.split (p is a compiled object)

```
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of',
'split', '']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

String	Class	Equivalent
\d	Decimal digit	[0-9]
\D	Non-digit	[^0-9]
\s	Any whitespace	[\t\n\r\f\v]
\S	Non-whitespace	[^ \t\n\r\f\v]
\w	Any alphanumeric	[a-zA-Z0-9_]
\W	Non alphanumeric	[^a-zA-Z0-9_]

Split

- Sometimes we also need to know the delimiters.
- Add parentheses in the RE!
- Compare the following calls:

```
>>> p = re.compile('\W+')
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2 = re.compile('(\W+)')
>>> p2.split('This... is a test.')
['This', '... ', 'is', ' ', 'a', ' ',
'test', '.', '']
```

Search and replace

- Find matches and replace them.
- Usage: `.sub(`*`replacement, string`*`[, `*`count=0`*`])`
- Returns a new string.
  - If the pattern is not found string is return unchanged
  - count: optional
    - Specifies the maximal number of replacements (when it is positive)

Search and replace - examples

```
>>> p = re.compile( '(blue|white|red)')
>>> p.sub( 'color', 'blue socks and red shoes')
'color socks and color shoes'
>>> p.sub( 'color', 'blue socks and red shoes', count=1)
'color socks and red shoes'

>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b--d-'
```

Empty matches are replaced only when they're not adjacent to a previous match:

```
>>> re.sub("a|x*",'-','abcd')
'--b-c-d-'
```

Naming groups

- Sometimes we use many groups
- Some of them should have meaningful names
- Syntax: (?P<name>…)
  - The '…' is where you need to write the actual regex

```
>>> p = re.compile(r'\W*(?P<word>\w+)\W*')
>>> m = p.search( '(((( Lots of punctuation )))' )
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
>>> m = p.finditer('(((( Lots of punctuation )))')
>>> for match in m:
...     print(match.group('word'))
...
Lots
of
punctuation
```

Backreferences

- Regex within regex
- Specify that the contents of an **earlier capturing group** must\can also be found at the current location in the string.
- \1 will succeed if the exact contents of group 1 can be found at the current position, and fails otherwise.
- Remember that Python’s string literals also use a backslash followed by numbers to allow including arbitrary characters in a string
  - **Be sure to use a raw strings!**

Example

- Explain this:

```
>>> p = re.compile(r'\W+(\w+)\W+\1')
>>> p.search('Paris in the the spring').group()
' the the'
```

Backreferences with names

- Syntax: use (?P=name) instead of \number
- In one regex do not use both numbered and named backreferences!

```
>>> p = re.compile(r'\W+(\w+)\W+\1')
>>> p.search('Paris in the the spring').group()
' the the'
>>> p = re.compile(r'(?P<word>\b\w+)\s+(?P=word)')
>>> p.search('Paris in the the spring').group()
'the the'
```

Pattern matching

A very sophisticated kind of logical test is to ask whether a string contains a ***pattern***

e.g. does a yeast promoter sequence contain the MCB binding site, ACGCGT?

20 bases upstream of the yeast gene YBR007C

```
name = 'YBR007C'
dna = 'TAATAAAAAACGCGTTGTCG'
if 'ACGCGT' in dna:
    print('%s has MCB!' % name)
```

YBR007C has MCB!

The membership operator **`in`**

The pattern for the MCB binding site

Regular expressions

We already defined a simple pattern: `ACGCGT`

What if we don't care about the 3rd position?

`ACGCGT` `ACCCGT` `ACACGT` `ACTCGT`

- Python provides a pattern-matching engine
- Patterns are called regular expressions
- They are extremely powerful
- Often called "regexps" for short
- module `re`

Motivation: N-glycosylation motif

- Common post-translational modification
- Attachment of a sugar group
- Occurs at asparagine residues with the consensus sequence NX1X2, where
  - X1 can be anything (but proline inhibits)
  - X2 is serine or threonine
- Can we detect potential N-glycosylation sites in a protein sequence?

Building regexps I: Character Groups

- In general square brackets denote a set of alternative possibilities
  - E.g. **`[abc]`** -> matches a,b, or c
- Use **`-`** to match a range of characters: `[A-Z]`
- Negation : **`[^X]`** matches anything but X
- **`.`** matches anything

Building regexps II: Abbreviations

- **`\d`** matches any decimal digits [0-9]
- **`\D`** matches any non-digit [^0-9]
- Equivalent syntax for …
  - whitespaces (**`\s`** and **`\S`**)
  - alphanumeric (**`\w`** and **`\W`**)

Building regexps III: Repetitions

- Use `*` to match none or any number of times
  - → E.g. ca`*`t matches: ct, cat, caat, caaat, caaaat, …
- Use `+` to match one or any number of times
  - → E.g. ca`+`t matches cat, caat, caaat, caaaat, …
- Use `?` to match none or once
  - → E.g. bio-`?`info matches bioinfo and bio-info
- Use `{m,n}` to specifically set the number of repetitions (min m, max n)
  - → E.g. ab`{1,3}`c will match abc, abbc, abbbc

Using regular expressions

- Compile a regular expression object (pattern) using `re.compile`
- pattern has a number of methods
  - → `match` (in case of success returns a Match object, otherwise None)
  - → `search` (scans through a string looking for a match)
  - → `findall` (returns a list of all matches)

```
>>> import re
>>> pattern = re.compile('[ACGT]')
>>> if pattern.match("A"): print("Matched")
Matched
>>> if pattern.match("a"): print("Matched")
>>>
```

successful match

unsuccessful, returns None by def. case sensitive

Matching alternative strings

- `/(this|that)/` matches "this" or "that"
- ...and is equivalent to: `/th(is|at)/`

case insensitive search pattern

```
>>> import re
>>> pattern=re.compile("(this|that|other)", re.IGNORECASE)
>>> pattern.search("Will match THIS") ## success
<re.Match object; span=(11, 15), match='THIS'>
>>> pattern.search("Will also match THat") ## success
<re.Match object; span=(16, 20), match='THat'>
>>> pattern.search("Will not match ot-her") ## will return None
>>>
```

Python returns a description of the match object

Word and string boundaries

^ matches the start of a string

$ matches the end of a string

\b matches word boundaries

"Escaping" special characters

- \ is used to "escape" characters that otherwise have meaning in a regexp
- so \[matches the character "["
  - if not escaped, "[" signifies the start of a list of alternative characters, as in [ACGT]
  - All special characters: `. ^ $ * + ? { [ ] \ | ( )`

Substitutions/Match Retrieval

- regexp methods can be used without compiling (less efficient but easier to use)
  - Example `re.sub` (substitution):

```
>>> re.sub("(red|blue|green)", "color", "blue socks and red shoes")
'color socks and color shoes'
```

matches one or more digits

```
>>> e,raw,frm,to = re.findall("\d+", \
... "E-value: 4, \
... Raw Bit Score: 165, \
... Match position: 362-419")
>>> print(e, raw, frm, to)
4 165 362 419
```

The result, a list of 4 strings, is assigned to 4 variables

\ allows multiple line commands alternatively, construct multi-line strings using triple quotes """ …"""

N-glycosylation site detector

```
import re

protein="MGMFFNLRSNIKKKAMDNGLSLPISRNGSSNNIKDKRSEHNSNSLKGKYRYQPRSTPSKFQLTVSITSLI \
IIAVLSLYLFISFLSGMGIGVSTQNGRSLLGSSKSSENYKTIDLEDEEYYDYDFEDIDPEVISKFDDGVQ \
HYLISQFGSEVLTPKDDEKYQRELNMLFDSTVEEYDLSNFEGAPNGLETRDHILLCIPLRNAADVLPLMF \
KHLMNLTYPHELIDLAFLVSDCSEGDTTLDALIAYSRHLQNGTLSQIFQEIDAVIDSQTKGTDKLYLKYM \
DEGYINRVHQAFSPPFHENYDKPFRSVQIFQKDFGQVIGQGFSDRHAVKVQGIRRKLMGRARNWLTANAL \
KPYHSWVYWRDADVELCPGSVIQDLMSKNYDVI".upper().replace("\n","")

for match in re.finditer("N[^P][ST]", protein):
    print(match.group(), match.span())
```

`re.finditer` provides an iterator over match-objects

`N[^P][ST]` - the main regular expression

```
> python3 nglycosylation.py
NGS (26, 29)
NLT (217, 220)
NGT (253, 256)
```

match.group and match.span print the actual matched string and the position-tuple.

Exercise

Modify the previous script such that it accepts a Fasta-formatted sequence via the CLI and returns location(s) and sequence(s) of putative N-glycosylation sites in this sequence. Use subroutines.

Desired behaviour:

```
> python3 nglycosylation.py P18892.fasta
>sp|P18892|BT1A1_BOVIN Butyrophilin subfamily 1 member A1 OS=Bos taurus GN=BTN1A1 PE=1
SV=2; length:526 bp
MAVFPNSCLAGCLLIFILLQLPKLDSAPFDVIGPQEPILAVVGEDAELPCRLSPNVSAKGMELRWFREKVSPAVFVSREGQEQEGEEMAE
YRGRVSLVEDHIAEGSVAVRIQEVKASDDGEYRCFFRQDENYEEAIVHLKVAALGSDPHISMKVQESGEIQLECTSVGWYPEPQVQWRTH
RGEEFPSMSESRNPDEEGLFTVRASVIIRDSSMKNVSCCIRNLLLGQEKEVEVSIPASFFPRLTPWMVAVAVILVVLGLLTIGSIFFTWR
LYKERSRQRRNEFSSKEKLLEELKWKRATLHAVDVTLDPDTAHPHLFLYEDSKSVRLEDSRQKLPEKPERFDSWPCVMGREAFTSGRHYW
EVEVGDRTDWAIGVCRENVMKKGFDPMTPENGFWAVELYGNGYWALTPLRTPLPLAGPPRRVGVFLDYESGDIFFYNMTDGSHIYTFSKA
SFSGPLRPFFCLWSCGKKPLTICPVTDGLEGVMVVADAKDISKEIPLSPMGEDSASGDIETLHSKLIPLQPSQGVP
Potential N-glycosylation sequence NVS at residue 55
Potential N-glycosylation sequence NVS at residue 215
Potential N-glycosylation sequence NMT at residue 437
```

Test your Regular Expressions

www.pythex.org

- Develop regular expressions
- Test them on examples of your choice

2REG		
	1osn	
		1VSN
9ins		

PDB IDs

([1-9][A-Za-z0-9]{3})

Summary

- **Regular expression** as powerful tools to match patterns
- Allow matching of character groups, repetitions, alternatives, etc.
- Learn the meaning of special characters: . ^ $ * + ? { [ ] \ | ( )
- Python offers **regexp** functions in the **re** module: match, search, findall, compile, etc.
- Regular expressions can be used to find motifs in sequences

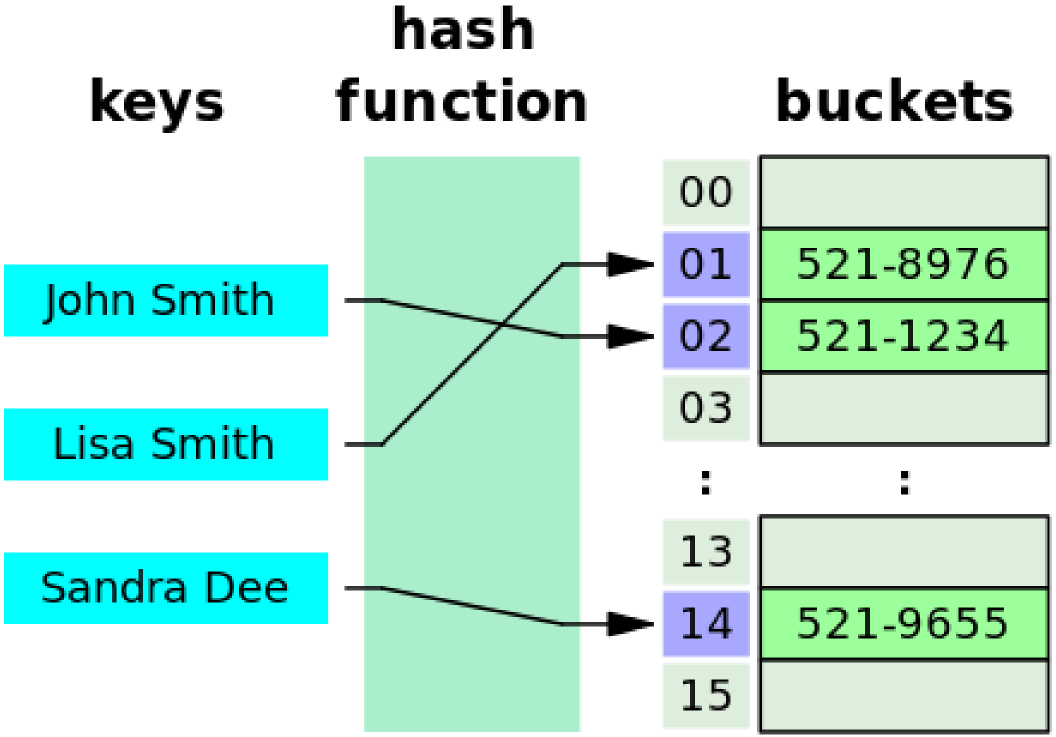
Dictionaries (hashes)

- Hash tables are efficient data structures for looking up values of specific entries
- Conveniently, Python provides this type of array called a dictionary
- A dictionary is a set of key:value pairs
  - like in a phone book

Squared brackets [ ] are used to index a dictionary

phone_book[“Donald Trump”] = “666-666666”

How does a dictionary work?



Keys and Values

- `keys` returns the list of keys in the hash
  - e.g. names, in the `name2seq` hash
- `values` returns the list of values
  - e.g. sequences, in the `name2seq` hash

Getting familiar with hashes

```
>>> phones={"Adrian":9063,\
... "Barry":9065,\
... "Theresa":9090 }
>>> phones.keys()
dict_keys(['Adrian', 'Barry', 'Theresa'])
>>> phones.values()
dict_values([9063, 9065, 9090])
>>> phones["Adrian"]
9063
>>> "Jeff" in phones
False
>>> phones["Jeff"] = 9999
>>> "Jeff" in phones
True
```

Creating an initial phone book

Asking for all keys

Asking for all values

Asking for a value, given a key

Checking whether a key is in the list

Inserting a single key:value pair

Checking whether a key is in the list

Looping through the dictionary

```
>>> for name in phones.keys():
...     print(name,phones[name])
...
```

Adrian 9063
Barry 9065
Theresa 9090
Jeff 9999

Application: read csv file into a hash

list.csv

```
Julian	Alaphilippe	100
Kasper	Asgreen	101
Dries	Devenyns	102
Yves	Lampaert	103
Enric	Mas	104
Michael	Mørkøv	105
Maximilian	Richeze	106
Elia	Viviani	107
```

read_file_to_hash.py

```
def read_file(filename):
    phonebook = {}
    with open(filename, "r") as f:
        for line in f.readlines()[1:]:
            name, lastname, number=line.strip().split("\t")
            phonebook[name]=number
        return phonebook

print( read_file("list.csv") )
```

```
> python3 read_file_to_hash.py
{'Kasper': '101', 'Dries': '102', 'Yves': '103', 'Enric': '104', 'Michael': '105', 'Maximilian': '106',
'Elia': '107'}
```

The genetic code as a hash

```
aa = {'ttt':'F', 'tct':'S', 'tat':'Y', 'tgt':'C',
      'ttc':'F', 'tcc':'S', 'tac':'Y', 'tgc':'C',
      'tta':'L', 'tca':'S', 'taa':'!', 'tga':'!',
      'ttg':'L', 'tcg':'S', 'tag':'!', 'tgg':'W',

      'ctt':'L', 'cct':'P', 'cat':'H', 'cgt':'R',
      'ctc':'L', 'ccc':'P', 'cac':'H', 'cgc':'R',
      'cta':'L', 'cca':'P', 'caa':'Q', 'cga':'R',
      'ctg':'L', 'ccg':'P', 'cag':'Q', 'cgg':'R',

      'att':'I', 'act':'T', 'aat':'N', 'agt':'S',
      'atc':'I', 'acc':'T', 'aac':'N', 'agc':'S',
      'ata':'I', 'aca':'T', 'aaa':'K', 'aga':'R',
      'atg':'M', 'acg':'T', 'aag':'K', 'agg':'R',

      'gtt':'V', 'gct':'A', 'gat':'D', 'ggt':'G',
      'gtc':'V', 'gcc':'A', 'gac':'D', 'ggc':'G',
      'gta':'V', 'gca':'A', 'gaa':'E', 'gga':'G',
      'gtg':'V', 'gcg':'A', 'gag':'E', 'ggg':'G' }
```

Application: Translating DNA into protein

dna_to_protein.py

```
import sys

def translate(dna):
    length = len(dna)
    if len(dna) % 3 != 0:
        print( "Warning: Length is not a multiple of 3!" )
        sys.exit()
    protein = ""
    i = 0
    while i < length:
        codon = dna[i:i+3]
        #print( codon )
        if not codon in aa:
            print( "Codon ",codon," is illegal" )
            sys.exit()
        protein += aa[codon]
        i+=3
    return protein

print( translate("gatgacgaaagttgt") )
#print( translate("gatgacgaaagttgta") )
print( translate("gatgacgiaagttgt") )
```

```
> python3 dna_to_protein.py
DDESC
Codon  gia  is illegal
```

Application: Counting residue frequencies

count_residues.py

```
def count_residues(seq):
    freq={}
    seq = seq.lower()
    for res in seq:
        if res in freq:
            freq[res]+=1
        else:
            freq[res]=1
    return freq

freq =
count_residues("gatgacgaaagttgt")
for residue in freq.keys():
    print(residue,":", freq[residue])
```

```
> python3 count_residues.py
g : 5
a : 5
t : 4
c : 1
```

Application: Counting N-mer frequencies

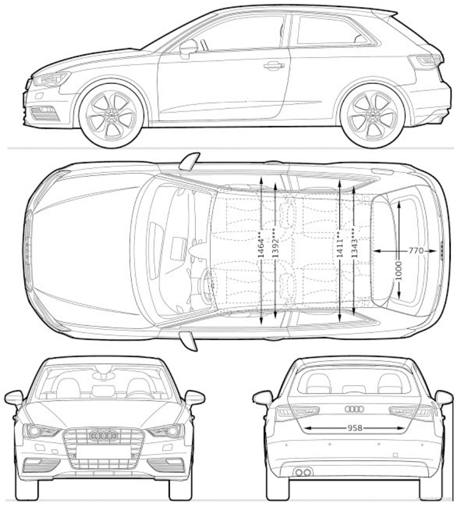
```
def count_nmers(seq, n):
    freq={}
    seq = seq.lower()
    for i in range(len(seq)-n+1):
        nmer=seq[i : i+n]
        if nmer in freq:
            freq[nmer]+=1 # incr. counter
        else: freq[nmer]=1 # first
occurence
    return freq

freq = count_nmers("gatgacgaaagttgt", 2)
for residue in freq.keys():
    print(residue,":", freq[residue])
```

```
> python3 count_nmers.py
ga : 3
at : 1
tg : 2
ac : 1
cg : 1
aa : 2
ag : 1
gt : 2
tt : 1
```

Classes

- A class is a collection of related variables and functions
- Object-oriented programming
- A class is a blueprint for objects
- E.g. a class for cars
  - → Properties: color, transmission, gears, type
  - → Methods: start, break, open_trunk, etc.



https://cgfrog.com/wp-content/uploads/2015/11/Audi-A3-2013-cgfrog_com_2.png

Class => “Blueprint” / Specifications

Properties

num_of_wheels
manufacturer
color
size
age

Functions

run()
stop()

Objects => “Real World” Instances

https://upload.wikimedia.org/wikipedia/commons/thumb/5/59/2013_Volkswagen_Golf_SE_BlueMotion_Technology_1.4_Front.jpg/2560px-2013_Volkswagen_Golf_SE_BlueMotion_Technology_1.4_Front.jpg

Properties

num_of_wheels = 4
manufacturer = “Volkswagen”
color = “grey”
size = “medium”
Age = 3

Functions

run()
stop()

https://en.wikipedia.org/wiki/File:2018_Ford_Escape_(ZG)_Trend_AWD_wagon_(2018-10-29)_01.jpg

Properties

num_of_wheels = 4
manufacturer = “Ford”
color = “white”
size = “medium”
Age = 3

Functions

run()
stop()

https://upload.wikimedia.org/wikipedia/commons/thumb/d/de/2017_Alfa_Romeo_Stelvio_Milano_Edizione_TD_Automatic_2.1.jpg/420px-2017_Alfa_Romeo_Stelvio_Milano_Edizione_TD_Automatic_2.1.jpg

Properties

num_of_wheels = 4
manufacturer = “Alfa Romeo”
color = “red”
size = “big”
Age = 1

Functions

run()
stop()

Syntax

car_class.py

```
# class definition
class Car:
    def __init__(self, manufacturer, color, size, age):
        num_of_wheels = 4
        self.manufacturer = manufacturer
        self.color = color
        self.size = size
        self.age = age

    def run(self):
        print("Brummmmm!")

    def stop(self):
        print("Eeeeeek!")

# class instances
alfa = Car("Alfa Romeo", "red", "medium", None)
mini = Car("Ford", "blue", "small", 3)

mini.run()
print(mini.color)
mini.age += 1
print(mini.age)
```

```
> python3 car_class.py
Brummmmm!
blue
4
```

Classes

```
class <classname>:
statement_1
.
.
statement_n
```

The methods of a class get the instance as the first parameter traditionally named **`self`**

The method **`__init__`** is called upon object construction (if available)

Classes

- Reminder: ***type*** = **data** representation + behaviour
- **Classes are <u>user-defined types</u>.**

```
class <classname>:
statement_1
.
.
statement_n
```

- **Objects** of a class are called **class instances**.

Example – multi_map

- A dictionary with more than one value for each key
- We could use something like:

```
>>> lst = d.get(key, [ ])
>>> lst.append(value)
>>> d[key] = lst
```

- We will now write a new **class** that will be a **wrapper** around a dict
- The class will have **methods** that allow us to keep **multiple values for each key**

multi_map.py

```
class multi_map:
    def __init__(self):
        '''Create an empty Multimap'''
        self.inner = dict()

    def get(self, key):
        '''Return list of values associated with key'''
        return self.inner.get(key, [])

    def put(self, key, value):
        '''Adds value to the list of values associated with key'''
        value_list = self.get(key)
        if value not in value_list:
            value_list.append(value)
            self.inner[key] = value_list

    def put_all(self, key, values):
        for v in values:
            self.put(key, v)

    def remove(self, key, value):
        value_list = self.get(key)
        if value in value_list:
            value_list.remove(value)
            self.inner[key] = value_list
            return True
        return False
```

```
m = multi_map()
m.put('Belgium', 'Brussels')
m.put('Belgium', 'Ghent')
m.put('Belgium', 'Antwerp')
m.put('Belgium', 'Bruges')
m.put('France', 'Paris')
m.put('France', 'Tours')
m.put_all('England',('London', 'Manchester', 'Moscow'))
m.remove('England', 'Moscow')
print(m.get('Belgium'))
print(m.get('England'))
```

```
> python multi_map.py
['Brussels', 'Ghent', 'Antwerp', 'Bruges']
['London', 'Manchester']
```

Example: Reverse complement

[a:b:c] "Count in increment of **c** starting at **a** inclusive up to **b** exclusive"

reverse_comp.py

```
import string

def revcomp(seq):
    translation = str.maketrans("agct", "tcga")
    comp = seq.translate(translation)
    rcomp = comp[::-1] # reversing comp
    return rcomp

dna = "cggcgt"
rev = revcomp(dna)
print("Revcomp of %s is %s"%(dna, rev))
```

```
> python3 reverse_comp.py
Revcomp of cggcgt is acgccg
```

string formatted with place holders

Revcomp in a DNA class

`self` refers to the current object, gives access to all its variables

reverse_comp_class.py

Class Constructor saves input sequence as object variable in lower case

```
import string

class DNA:
    def __init__(self, sequence):
        self.seq = sequence.lower()

    def revcomp(self):
        translation = str.maketrans("agct", "tcga")
        comp = self.seq.translate(translation)
        self.revcomp = comp[::-1]

    def report(self):
        print ("Revcomp of %s is %s"% (self.seq,self.revcomp))

dna = DNA("accggcatg")# Creating a DNA object
dna.revcomp()
dna.report()
```

Method Calls

Useful to structure code :
add additional DNA sequence functionality to this class
-> e.g. a function that calculates GC-contents, translation to protein etc.

Everything is an object

- Instances of lists, strings, etc. are objects with built-in methods
- Explore available methods using `dir`:

```
>>> dir("hello")
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> help("hello".count)
Help on built-in function count:

count(...) method of builtins.str instance
    S.count(sub[, start[, end]]) -> int

    Return the number of non-overlapping occurrences of substring sub in
    string S[start:end].  Optional arguments start and end are
    interpreted as in slice notation.
>>> "hello".count("l")
2
```

String object Method

. (dot) applies method to object

Summary

- Use dictionaries if you need to lookup your values frequently in a program
- Dictionaries can store key-value pairs in a very efficient way using hash tables
- Use classes to organize related variables and methods into a shared structure
- Classes are blueprints for objects
- Everything in Python is an object