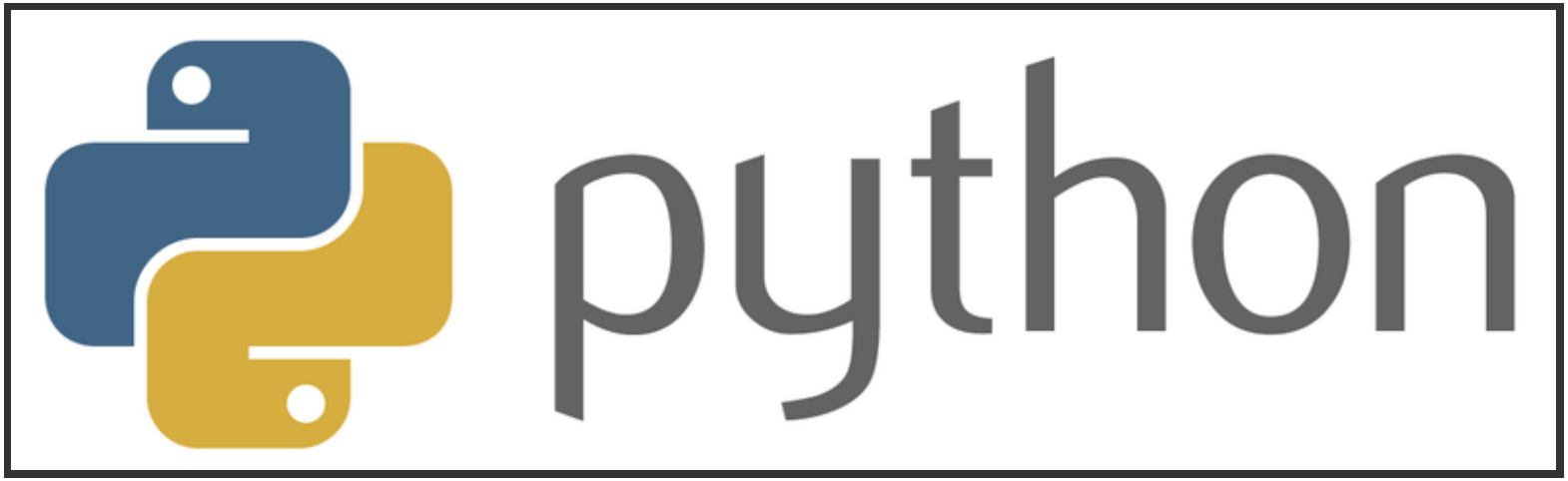
Introduction to



with Application to Bioinformatics

- Day 5

Sharing code

Share code snippet

- Pastebin

Collaboration space for notebooks

- Colab

Share codebase - Advanced option

- Github

Review

- Lists
  - Create a list named letters_list containing the elements `'a'`, `'b'`, `'c'`.
  - Reverse the list letters_list

In [5]:

```
# Create a list containing the elements `'a'`, `'b'`, `'c'`
letters_list = ['a', 'b', 'c']
print(letters_list)
```

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

In [31]:

```
# Reverse it
letters_list.reverse()
print(letters_list)
```

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

In [8]:

```
# Create a dictionary containing the keys a and b. Both should have the value 1
letters_dict = {'a': 1, 'b': 1}
print(letters_dict)
```

```
{'a': 1, 'b': 1}
```

In [9]:

```
# Change the value of a to 2
letters_dict['a'] = 2
```

```
print(letters_dict)
```

```
{'a': 2, 'b': 1}
```

In [16]:

```
# Set the variable `title` to `"A movie"` and `rating` to 10.
title = 'A movie'
rating = 10
```

In [18]:

```
# Use formatting to produce: "The movie A movie got rating 10!"
print("The movie {0} got rating {1}!".format(title, rating))
print(f"The movie {title} got rating {rating}!")
```

```
The movie A movie got rating 10!
The movie A movie got rating 10!
```

In [33]:

```
# Functions and keyword arguments
# def open(file, mode='r', buffering=- 1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
#fh = open('aFile.txt', 'r', 'utf-8')
#fh = open('Day_4.py', 'r', encoding='utf-8')
```

TODAY

- review
- regex
- sumup

Review Day 4

- More control!
    - variables scope
    - `None`
    - keyword arguments
    - documentation, comments...
- Pandas

```python
my_list = ['Initial element 1', 'Initial element 2']

def function_returning_values():
    return ['Function element 1', 'Function element 2']

my_list = function_returning_values()
print(my_list)
```

```
['Function element 1', 'Function element 2']
```

```python
my_list = ['Initial element 1', 'Initial element 2']

def function_returning_values():
    return ['Function element 1', 'Function element 2']

my_list = function_returning_values()
print(my_list)
```

```
['Function element 1', 'Function element 2']
```

In [35]:

```python
my_list = ['Initial element 1', 'Initial element 2']

def function_returning_values():
    my_list = ['Function element 1', 'Function element 2']

function_returning_values()
print(my_list)
```

```
['Initial element 1', 'Initial element 2']
```

```python
my_list = ['Initial element 1', 'Initial element 2']

def function_returning_values():
    return ['Function element 1', 'Function element 2']

my_list = function_returning_values()
print(my_list)
```

```
['Function element 1', 'Function element 2']
```

In [35]:

```python
my_list = ['Initial element 1', 'Initial element 2']

def function_returning_values():
    my_list = ['Function element 1', 'Function element 2']

function_returning_values()
print(my_list)
```

```
['Initial element 1', 'Initial element 2']
```

In [36]:

```python
my_list = ['Initial element 1', 'Initial element 2']

def function_returning_values():
    my_list = ['Function element 1', 'Function element 2']

my_list = function_returning_values()
print(my_list)
```

```
None
```

```python
# `None` means "nothing". Use it to check your variables

variable = None
if variable:
    print('if variable')
if not variable:
    print('if not variable')
if variable is not None:
    print('if variable is not None')
if variable is None:
    print('if variable is None')
```

```
if not variable
if variable is None
```

Keyword arguments

- Defining function

```
def open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True,
opener=None)
```

- Calling the function

```
open(filename,
encoding="utf-8")
```

Documentation and getting help

- `help(sys)` for a module
- `help(math.sqrt)` for a function

Documentation and getting help

- `help(sys)` for a module
- `help(math.sqrt)` for a function
- write comments `# why do I do this?`
- write documentation `"""what is this? how do you use it?"""`

Writing readable code

Writing readable code

```
def f(a, b):
    for c in open(a):
        if c.startswith(b):
            print(c)
```

Writing readable code

```
def f(a, b):
    for c in open(a):
        if c.startswith(b):
            print(c)
```

==>

```
def print_lines(filename, start):
    """Print all lines in the file that starts with the given string."""
    for line in open(filename):
        if line.startswith(start):
            print(line)
```

Writing readable code

```
def f(a, b):
    for c in open(a):
        if c.startswith(b):
            print(c)
```

==>

```
def print_lines(filename, start):
    """Print all lines in the file that starts with the given string."""
    for line in open(filename):
        if line.startswith(start):
            print(line)
```

Care about the names of your variables and functions

Pandas

- Read files

```
dataframe = pandas.read_table('mydata.txt', sep='|',
index_col=0)
dataframe = pandas.read_csv('mydata.csv')
```

- Select rows and colums

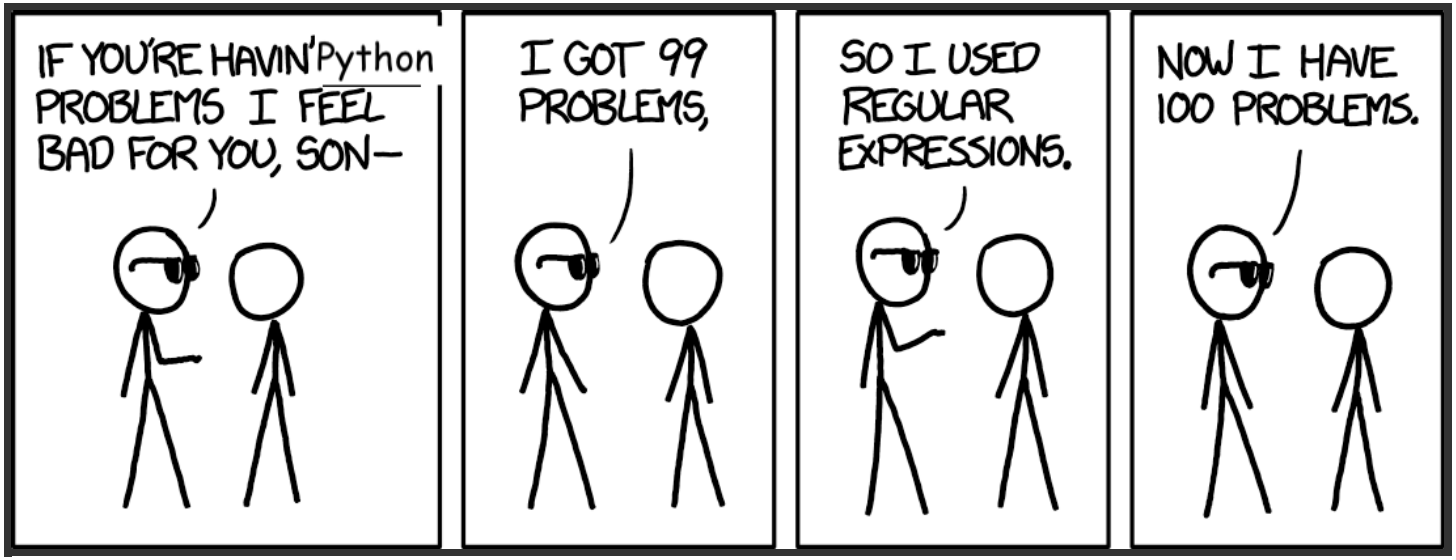
```
dataframe.columname
dataframe.loc[rowname]
dataframe.loc[dataframe.age == 20 ]
```

- Plot it

```
dataframe.plot(kind='line', x='column1', y='column2')
```

TODAY

- Regular expressions



- Sum up of the course

Regular Expressions

- A smarter way of searching text
- search&replace
- Relatively advanced topic

Regular Expressions

Regular Expressions

- A formal language for defining search patterns

Regular Expressions

- A formal language for defining search patterns
- Enables to search not only for exact strings but controlled variations of that string.

Regular Expressions

- A formal language for defining search patterns
- Enables to search not only for exact strings but controlled variations of that string.
- Why?

Regular Expressions

- A formal language for defining search patterns
- Enables to search not only for exact strings but controlled variations of that string.
- Why?
- Examples:
    - Find variations in a protein or DNA sequence
        - `"MVR???A"`
        - `"ATG???TAG"`
    - American/British spelling, endings and other variants:
        - salpeter, salpetre, saltpeter, nitre, niter or KNO3
        - hemaglobin, heamoglobin, hemaglobins, heamoglobin's
        - catalyze, catalyse, catalyzed...
    - A pattern in a vcf file
        - a digit appearing after a tab

Regular Expressions

Regular Expressions

- When?

Regular Expressions

- When?
- To find information
    - in your `vcf` or `fasta` files
    - in your code
    - in your next essay
    - in a database
    - online
    - in a bunch of articles
    - ...

Regular Expressions

- When?
- To find information
    - in your `vcf` or `fasta` files
    - in your code
    - in your next essay
    - in a database
    - online
    - in a bunch of articles
    - ...
- Search/replace
    - becuase → because
    - color → colour
    - `\t` (tab) → `"    "` (four spaces)

Regular Expressions

- When?
- To find information
    - in your `vcf` or `fasta` files
    - in your code
    - in your next essay
    - in a database
    - online
    - in a bunch of articles
    - ...
- Search/replace
    - becuase → because
    - color → colour
    - `\t` (tab) → `"    "` (four spaces)
- Supported by most programming languages, text editors, search engines...

Defining a search pattern

color
colour
colours
coloring
coloured

```
col[ou]+r.*
```

salpeter
salpetre
saltpeter

```
salt?pet(er|re)
```

Common operations

Building blocks for creating patterns

- `.` matches any character (once)
- `?` repeat previous pattern 0 or 1 times
- `*` repeat previous pattern 0 or more times
- `+` repeat previous pattern 1 or more times

Common operations

Building blocks for creating patterns

- `.` matches any character (once)
- `?` repeat previous pattern 0 or 1 times
- `*` repeat previous pattern 0 or more times
- `+` repeat previous pattern 1 or more times

Pattern for matching the colour family

```
colour.*
```

Common operations

Building blocks for creating patterns

- `.` matches any character (once)
- `?` repeat previous pattern 0 or 1 times
- `*` repeat previous pattern 0 or more times
- `+` repeat previous pattern 1 or more times

Pattern for matching the colour family

```
colour.*
```

`.*` matches everything (including the empty string)!

Common operations

Building blocks for creating patterns

- `.` matches any character (once)
- `?` repeat previous pattern 0 or 1 times
- `*` repeat previous pattern 0 or more times
- `+` repeat previous pattern 1 or more times

Pattern for matching the colour family

```
colour.*
```

`.*` matches everything (including the empty string)!

Pattern for matching the different spellings

```
salt?peter
```

Common operations

Building blocks for creating patterns

- `.` matches any character (once)
- `?` repeat previous pattern 0 or 1 times
- `*` repeat previous pattern 0 or more times
- `+` repeat previous pattern 1 or more times

Pattern for matching the colour family

```
colour.*
```

`.*` matches everything (including the empty string)!

Pattern for matching the different spellings

```
salt?peter
```

What about the different endings: er-re?

`"salt?pet.."`

Common operations

Building blocks for creating patterns

- `.` matches any character (once)
- `?` repeat previous pattern 0 or 1 times
- `*` repeat previous pattern 0 or more times
- `+` repeat previous pattern 1 or more times

Pattern for matching the colour family

```
colour.*
```

`.*` matches everything (including the empty string)!

Pattern for matching the different spellings

```
salt?peter
```

What about the different endings: er-re?

`"salt?pet.."`

saltpeter

"saltpet88"

"salpetin"

"saltpet "

More common operations - classes of characters

- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace

More common operations - classes of characters

- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace

`\w+`

```
def functionName(arg1, arg2, arg3):
    final_value = 0
    # comments
    return final_value
```

More common operations - classes of characters

- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace

`\d+`

```
def functionName(arg1, arg2, arg3):
    final_value = 0
    # comments
    return final_value
```

More common operations - classes of characters

- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace

`\s+`

```
def functionName(arg1, arg2, arg3):
    final_value = 0
    # comments
    return final_value
```

More common operations - classes of characters

- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace
- `[abc]` matches a single character defined in this set {a, b, c}
- `[^abc]` matches a single character that is **not** a, b or c

More common operations - classes of characters

- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace
- `[abc]` matches a single character defined in this set {a, b, c}
- `[^abc]` matches a single character that is **not** a, b or c

`[a-z]` matches all letters between `a` and `z` (the english alphabet).

`[a-z]+` matches any (lowercased) english word.

More common operations - classes of characters

- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace
- `[abc]` matches a single character defined in this set {a, b, c}
- `[^abc]` matches a single character that is **not** a, b or c

`[a-z]` matches all letters between `a` and `z` (the english alphabet).

`[a-z]+` matches any (lowercased) english word.

`salt?pet[er]+`

saltpeter

salpetre

~~"saltpet88"~~

~~"salpetin"~~

~~"saltpet "~~

Example - finding patterns in vcf

```
1   920760  rs80259304  T   C   .   PASS    AA=T;AC=18;AN=120;
DP=190;GP=1:930897;BN=131 GT:DP:CB    0/1:1:SM 0/0:4/SM...
```

Example - finding patterns in vcf

```
1   920760  rs80259304  T   C   .   PASS    AA=T;AC=18;AN=120;
DP=190;GP=1:930897;BN=131 GT:DP:CB    0/1:1:SM 0/0:4/SM...
```

- Find a sample:

```
0/0 0/1 1/1 ...
```

Example - finding patterns in vcf

```
1   920760  rs80259304  T   C   .   PASS    AA=T;AC=18;AN=120;
DP=190;GP=1:930897;BN=131 GT:DP:CB    0/1:1:SM 0/0:4/SM...
```

- Find a sample:

`0/0 0/1 1/1` ...

`"[01]/[01]"` (or `"\d/\d"`)

Example - finding patterns in vcf

```
1   920760  rs80259304  T   C   .   PASS    AA=T;AC=18;AN=120;
DP=190;GP=1:930897;BN=131 GT:DP:CB    0/1:1:SM 0/0:4/SM...
```

- Find a sample:

`0/0 0/1 1/1` ...

`"[01]/[01]"` (or `"\d/\d"`)

`\s[01]/[01]:`

Example - finding patterns in vcf

```
1   920760  rs80259304  T   C   .   PASS    AA=T;AC=18;AN=120;
DP=190;GP=1:930897;BN=131 GT:DP:CB    0/1:1:SM 0/0:4/SM...
```

- Find all lines containing more than one homozygous sample.

Example - finding patterns in vcf

```
1   920760  rs80259304  T   C   .   PASS    AA=T;AC=18;AN=120;
DP=190;GP=1:930897;BN=131 GT:DP:CB    0/1:1:SM 0/0:4/SM...
```

- Find all lines containing more than one homozygous sample.

```
... 1/1:...  ... 1/1:...  ...
```

Example - finding patterns in vcf

```
1   920760  rs80259304  T   C   .   PASS    AA=T;AC=18;AN=120;
DP=190;GP=1:930897;BN=131 GT:DP:CB    0/1:1:SM 0/0:4/SM...
```

- Find all lines containing more than one homozygous sample.

```
... 1/1:...  ... 1/1:...  ...
```

```
.*1/1.*1/1.*
```

Example - finding patterns in vcf

```
1   920760  rs80259304  T   C   .   PASS    AA=T;AC=18;AN=120;
DP=190;GP=1:930897;BN=131 GT:DP:CB    0/1:1:SM 0/0:4/SM...
```

- Find all lines containing more than one homozygous sample.

```
... 1/1:...  ... 1/1:...  ...
```

```
.*1/1.*1/1.*
```

```
.*\s1/1:.*\s1/1:.*
```

Exercise 1

- `.` matches any character (once)
- `?` repeat previous pattern 0 or 1 times
- `*` repeat previous pattern 0 or more times
- `+` repeat previous pattern 1 or more times
- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace
- `[abc]` matches a single character defined in this set {a, b, c}
- `[^abc]` matches a single character that is **not** a, b or c
- `[a-z]` matches any (lowercased) letter from the english alphabet

- `.*` matches anything

- https://regex101.com/

→ Notebook Day_5_Exercise_1 (~30 minutes)

Regular expressions in Python

Regular expressions in Python

In [37]:

```
# Import module
import re
```

Regular expressions in Python

In [37]:

```
# Import module
import re
```

In [38]:

```
# Define a pattern
p = re.compile('ab*')
p
```

Out[38]:

```
re.compile(r'ab*', re.UNICODE)
```

Searching

Searching

In [41]:

```
# Search pattern in string
p = re.compile('ab*')

p.search('abc')
```

Out[41]:

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

Searching

In [41]:

```
# Search pattern in string
p = re.compile('ab*')

p.search('abc')
```

Out[41]:

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

In [42]:

```
print(p.search('cb'))
```

```
None
```

Searching

In [41]:

```
# Search pattern in string
p = re.compile('ab*')

p.search('abc')
```

Out[41]:

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

In [42]:

```
print(p.search('cb'))
```

```
None
```

In [43]:

```
p = re.compile('HELLO')
m = p.search('gsdfgsdfgs  HELLO  __!@£§≈[|ÅÄÖ,…’fi]')

print(m)
```

```
<re.Match object; span=(12, 17), match='HELLO'>
```

Case insensitiveness

In [46]:

```
# Remember, [a-z]+ matches any lower case english word
p = re.compile('[a-z]+')
result = p.search('ATGAAA')
print(result)
```

```
None
```

Case insensitiveness

In [46]:

```
# Remember, [a-z]+ matches any lower case english word
p = re.compile('[a-z]+')
result = p.search('ATGAAA')
print(result)
```

```
None
```

In [49]:

```
p = re.compile('[a-z]+', re.IGNORECASE)

result = p.search('ATGAAA')
result
```

Out[49]:

```
<re.Match object; span=(0, 6), match='ATGAAA'>
```

The match object

The match object

In [51]:

```
p = re.compile('[a-z]+', re.IGNORECASE)

result = p.search('123 ATGAAA 456')
result
```

Out[51]:

```
<re.Match object; span=(4, 10), match='ATGAAA'>
```

The match object

In [51]:

```
p = re.compile('[a-z]+', re.IGNORECASE)

result = p.search('123 ATGAAA 456')
result
```

Out[51]:

```
<re.Match object; span=(4, 10), match='ATGAAA'>
```

`result.group()` : Return the string matched by the expression

`result.start()` : Return the starting position of the match

`result.end()` : Return the ending position of the match

`result.span()` : Return both (start, end)

The match object

In [51]:

```
p = re.compile('[a-z]+', re.IGNORECASE)

result = p.search('123 ATGAAA 456')
result
```

Out[51]:

```
<re.Match object; span=(4, 10), match='ATGAAA'>
```

`result.group()` : Return the string matched by the expression

`result.start()` : Return the starting position of the match

`result.end()` : Return the ending position of the match

`result.span()` : Return both (start, end)

In [52]:

```
result.group()
```

Out[52]:

```
'ATGAAA'
```

The match object

In [51]:

```
p = re.compile('[a-z]+', re.IGNORECASE)

result = p.search('123 ATGAAA 456')
result
```

Out[51]:

```
<re.Match object; span=(4, 10), match='ATGAAA'>
```

`result.group()` : Return the string matched by the expression

`result.start()` : Return the starting position of the match

`result.end()` : Return the ending position of the match

`result.span()` : Return both (start, end)

In [52]:

```
result.group()
```

Out[52]:

```
'ATGAAA'
```

In [53]:

```
result.start()
```

Out[53]:

```
4
```

In [54]:

```
result.end()
```

Out[54]:

```
10
```

In [55]:

```
result.span()
```

Out[55]:

```
(4, 10)
```

Zero or more...?

In [56]:

```
p = re.compile('.*HELLO.*')
```

Zero or more...?

In [56]:

```
p = re.compile('.*HELLO.*')
```

In [57]:

```
m = p.search('lots of text  HELLO  more text and characters!!! ^^')
```

Zero or more...?

In [56]:

```
p = re.compile('.*HELLO.*')
```

In [57]:

```
m = p.search('lots of text  HELLO  more text and characters!!! ^^')
```

In [58]:

```
m.group()
```

Out[58]:

```
'lots of text  HELLO  more text and characters!!! ^^'
```

Zero or more...?

In [56]:

```
p = re.compile('.*HELLO.*')
```

In [57]:

```
m = p.search('lots of text  HELLO  more text and characters!!! ^^')
```

In [58]:

```
m.group()
```

Out[58]:

```
'lots of text  HELLO  more text and characters!!! ^^'
```

The $*$ is **greedy**.

Finding all the matching patterns

In [71]:

```
# Find all instance of the defined pattern
p = re.compile('HELLO')
matches = p.finditer('lots of text  HELLO  more text  HELLO ... and characters!!! ^^')
print(matches)
```

```
<callable_iterator object at 0x7ff202b6efa0>
```

Finding all the matching patterns

In [71]:

```
# Find all instance of the defined pattern
p = re.compile('HELLO')
matches = p.finditer('lots of text  HELLO  more text  HELLO ... and characters!!! ^^')
print(matches)
```

```
<callable_iterator object at 0x7ff202b6efa0>
```

In [72]:

```
# Loop through matches
for match in matches:
    print(f'Found {match.group()} at position {match.start()}')
```

```
Found HELLO at position 14
Found HELLO at position 32
```

How to find a full stop?

In [79]:

```
txt = "The first full stop is here: ."
pattern = re.compile('.')

match = pattern.search(txt)
print('"{}" at position {}'.format(match.group(), match.start()))
```

```
"T" at position 0
```

In [85]:

```
# Print all matches
matches = p.finditer(txt)
#for match in matches:
#    print('"{}" at position {}'.format(match.group(), match.start()))
```

How to find a full stop?

In [79]:

```
txt = "The first full stop is here: ."
pattern = re.compile('.')

match = pattern.search(txt)
print('"{}" at position {}'.format(match.group(), match.start()))
```

```
"T" at position 0
```

In [85]:

```
# Print all matches
matches = p.finditer(txt)
#for match in matches:
#    print('"{}" at position {}'.format(match.group(), match.start()))
```

In [86]:

```
# Use escape character to search
p = re.compile('\.')

m = p.search(txt)
print('"{}" at position {}'.format(m.group(), m.start()))
```

```
"." at position 29
```

More operations

- `\` escaping a character
- `^` beginning of the string
- `$` end of string
- `|` boolean `or`

More operations

- `\` escaping a character
- `^` beginning of the string
- `$` end of string
- `|` boolean `or`

```
^hello$
```

More operations

- `\` escaping a character
- `^` beginning of the string
- `$` end of string
- `|` boolean `or`

```
^hello$
```

```
salt?pet(er|re) | nit(er|re) | KNO3
```

Substitution

Finally, we can fix our spelling mistakes!

In [87]:

```
txt = "Do it   becuase   I say so,     not becuase you want!"
```

Substitution

Finally, we can fix our spelling mistakes!

In [87]:

```
txt = "Do it   becuase   I say so,     not becuase you want!"
```

In [89]:

```
# Spell the word because correctly
import re
p = re.compile('becuase')
txt = p.sub('because', txt)
print(txt)
```

```
Do it   because   I say so,     not because you want!
```

Substitution

Finally, we can fix our spelling mistakes!

In [87]:

```
txt = "Do it   becuase   I say so,     not becuase you want!"
```

In [89]:

```
# Spell the word because correctly
import re
p = re.compile('becuase')
txt = p.sub('because', txt)
print(txt)
```

```
Do it   because   I say so,     not because you want!
```

In [90]:

```
# Remove additional spaces
p = re.compile('\s+')
p.sub(' ', txt)
```

Out[90]:

```
'Do it because I say so, not because you want!'
```

Overview

- Construct regular expressions

```
p = re.compile()
```

- Searching

```
p.search(text)
```

- Substitution

```
p.sub(replacement, text)
```

Typical code structure:

```
pattern = re.compile( ... )
match = pattern.search('string goes here')
if m:
    print('Match found: ', match.group())
else:
    print('No match')
```

Regular expressions

- A powerful tool to search and modify text
- There is much more to read in the docs
- Note: regex comes in different flavours. If you use it outside Python, there might be small variations in the syntax.

Exercise 2

- `.` matches any character (once)
- `?` repeat previous pattern 0 or 1 times
- `*` repeat previous pattern 0 or more times
- `+` repeat previous pattern 1 or more times
- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace
- `[abc]` matches a single character defined in this set {a, b, c}
- `[^abc]` matches a single character that is **not** a, b or c
- `[a-z]` matches any (lowercased) letter from the english alphabet
- `.*` matches anything
- `\` escaping a character
- `^` beginning of the string
- `$` end of string
- `|` boolean `or`

Read more: full documentation https://docs.python.org/3.9/library/re.html
→ Notebook Day_5_Exercise_2 (~30 minutes)

Sum up!

Processing files - looping through the lines

```
fh = open('myfile.txt')
for line in fh:
    do_stuff(line)
```

Store values

```
iterations = 0
information = []

fh = open('myfile.txt', 'r')
for line in fh:
    iterations += 1
    information += do_stuff(line)
```

Values

- Base types:

```
- str     "hello"
- int     5
- float   5.2
- bool    True
```

- Collections:

```
- list  ["a", "b", "c"]
- dict  {"a": "alligator", "b": "bear", "c":
"cat"}
- tuple ("this", "that")
- set   {"drama", "sci-fi"}
```

Assign values

```
iterations = 0
score = 5.2
```

Compare and membership

```
+, -, *,...    # mathematical
and, or, not   # logical
==, !=         # (in)equality
<, >, <=, >=   # comparison
in             # membership
```

```python
value = 4
nextvalue = 1
nextvalue += value
print('nextvalue: ', nextvalue, 'value: ', value)
```

```
nextvalue:  5 value:  4
```

```
value = 4
nextvalue = 1
nextvalue += value
print('nextvalue: ', nextvalue, 'value: ', value)
```

```
nextvalue:  5 value:  4
```

In [40]:

```
x = 5
y = 7
z = 2
x > 6 and y == 7 or z > 1
```

Out[40]:

```
True
```

```
value = 4
nextvalue = 1
nextvalue += value
print('nextvalue: ', nextvalue, 'value: ', value)
```

```
nextvalue:  5 value:  4
```

In [40]:

```
x = 5
y = 7
z = 2
x > 6 and y == 7 or z > 1
```

Out[40]:

```
True
```

In [41]:

```
(x > 6 and y == 7) or z > 1
```

Out[41]:

```
True
```

Strings

- Works like a list of characters
  - define using " or '
  - ```
    s += "more words"  # add content
    ```
  - ```
    s[4]               # get character at index 4
    ```
  - ```
    'e' in s           # check for membership
    ```
  - ```
    len(s)             # check size
    ```

Strings

- Works like a list of characters
  - define using " or '
  - ```
    s += "more words"  # add content
    ```
  - ```
    s[4]               # get character at
    index 4
    ```
  - ```
    'e' in s           # check for membership
    ```
  - ```
    len(s)             # check size
    ```
- But are immutable
  - ```
    > s[2] = 'i'
    ```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item
assignment
```
```

Strings

Raw text

- Common manipulations:
  - ```
    s.strip()  # remove unwanted spacing
    ```
  - ```
    s.split()  # split line into columns
    ```
  - ```
    s.upper(), s.lower()  # change the case
    ```
    ```

Strings

Raw text

- Common manipulations:
  - ```
    s.strip()  # remove unwanted spacing
    ```
  - ```
    s.split()  # split line into columns
    ```
  - ```
    s.upper(), s.lower()  # change the case
    ```
- Regular expressions help you find and replace strings.
  - ```
    p = re.compile('A.A.A')
    p.search(dnastring)
    ```
  - ```
    p = re.compile('T')
    p.sub('U', dnastring)
    ```
    ```

```python
import re

p = re.compile('p.*\sp')  # the greedy star!

p.search('a python programmer writes python code').group()
```

Out[92]:

```
'python programmer writes p'
```

Collections

Can contain strings, integer, booleans...

- **Mutable**: you can *add*, *remove*, *change* values
  - Lists:

    ```
    mylist.append('value')
    ```

  - Dicts:

    ```
    mydict['key'] = 'value'
    ```

  - Sets:

    ```
    myset.add('value')
    ```

Collections

- Test for membership:

```
value in myobj
```

- Check size:

```
len(myobj)
```

Lists

- Ordered!

```
todolist = ["work", "sleep", "eat", "work"]

todolist.sort()
todolist.reverse()
todolist[2]
todolist[-1]
todolist[2:6]
```

```
todolist = ["work", "sleep", "eat", "work"]
```

In [94]:

```
todolist.sort()
print(todolist)
```

```
['eat', 'sleep', 'work', 'work']
```

In [95]:

```
todolist.reverse()
print(todolist)
```

```
['work', 'work', 'sleep', 'eat']
```

In [96]:

```
todolist[2]
```

Out[96]:

```
'sleep'
```

In [99]:

```
todolist[-1]
```

Out[99]:

```
'eat'
```

In [103]:

```
todolist[2:]
```

Out[103]:

```
['eat', 'work']
```

Dictionaries

- Keys have values

```
mydict = {"a": "alligator", "b": "bear", "c": "cat"}
counter = {"cats": 55, "dogs": 8}

mydict["a"]
mydict.keys()
mydict.values()
```

```python
counter = {'cats': 0, 'others': 0}

for animal in ['zebra', 'cat', 'dog', 'cat']:
    if animal == 'cat':
        counter['cats'] += 1
    else:
        counter['others'] += 1

counter
```

Out[104]:

```
{'cats': 2, 'others': 2}
```

Sets

- Bag of values
    - No order
    - No duplicates
    - Fast membership checks
    - Logical set operations (union, difference, intersection...)

```
myset = {"drama", "sci-fi"}

myset.add("comedy")

myset.remove("drama")
```

```python
todolist = ["work", "sleep", "eat", "work"]

todo_items = set(todolist)
todo_items
```

Out[105]:

```
{'eat', 'sleep', 'work'}
```

```
todolist = ["work", "sleep", "eat", "work"]

todo_items = set(todolist)
todo_items
```

Out[105]:

```
{'eat', 'sleep', 'work'}
```

In [106]:

```
todo_items.add("study")
todo_items
```

Out[106]:

```
{'eat', 'sleep', 'study', 'work'}
```

```
todolist = ["work", "sleep", "eat", "work"]

todo_items = set(todolist)
todo_items
```

Out[105]:

```
{'eat', 'sleep', 'work'}
```

In [106]:

```
todo_items.add("study")
todo_items
```

Out[106]:

```
{'eat', 'sleep', 'study', 'work'}
```

In [107]:

```
todo_items.add("eat")
todo_items
```

Out[107]:

```
{'eat', 'sleep', 'study', 'work'}
```

Tuples

- A group (usually two) of values that belong together
  - ```
    tup = (max_length, sequence)
    ```
  - An ordered sequence (like lists)
  - ```
    length = tup[0]  # get content at index 0
    ```
  - Immutable

Tuples

- A group (usually two) of values that belong together
  - ```
    tup = (max_length, sequence)
    ```
  - An ordered sequence (like lists)
  - ```
    length = tup[0]  # get content at index 0
    ```
  - Immutable

In [53]:

```
tup = (2, 'xy')
tup[0]
```

Out[53]:

```
2
```

In [54]:

```
tup[0] = 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-54-874559a0c62a> in <module>
```

```
----> 1 tup[0] = 2

TypeError: 'tuple' object does not support item assignm
ent
```

Tuples in functions

```
def find_longest_seq(file):
    # some code here...
    return length, sequence
```

Tuples in functions

```python
def find_longest_seq(file):
    # some code here...
    return length, sequence
```

```python
answer = find_longest_seq(filepath)
print('length', answer[0])
print('sequence', answer[1])
```

Tuples in functions

```python
def find_longest_seq(file):
    # some code here...
    return length, sequence
```

```python
answer = find_longest_seq(filepath)
print('length', answer[0])
print('sequence', answer[1])
```

```python
answer = find_longest_seq(filepath) # return as a tuple
length, sequence = find_longest_seq(filepath) # return as two variables
```

Deciding what to do

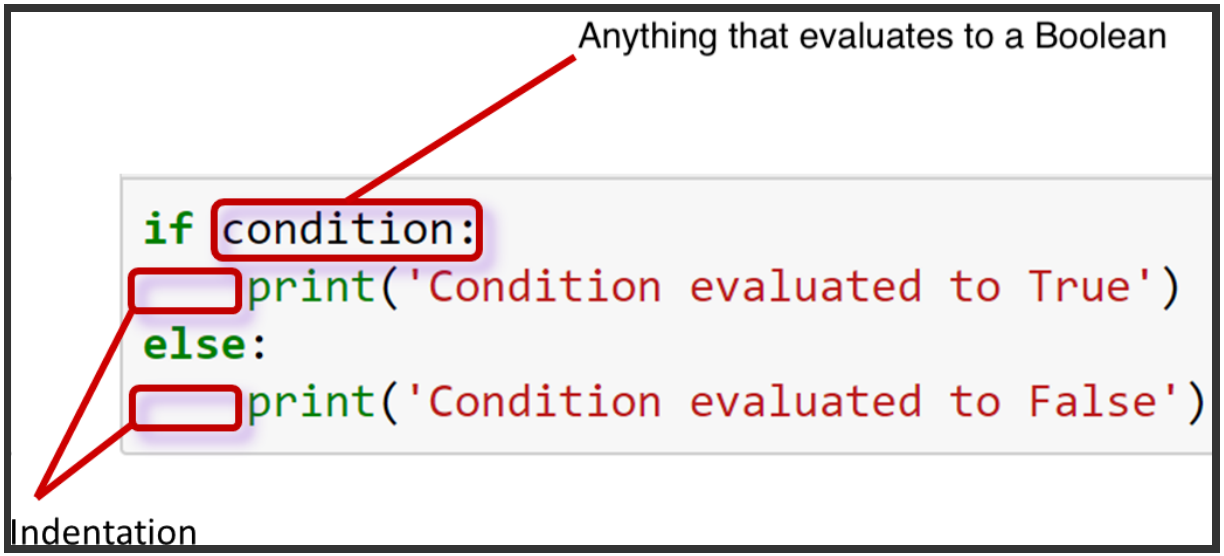
```
if count > 10:
   print('big')
elif count > 5:
   print('medium')
else:
   print('small')
```

```python
shopping_list = ['bread', 'egg', ' butter', 'milk']
tired         = True

if len(shopping_list) > 4:
    print('Really need to go shopping!')
elif not tired:
    print('Not tired? Then go shopping!')
else:
    print('Better to stay at home')
```

```
Better to stay at home
```

Deciding what to do - if statement



Program flow - for loops

```
information = []
fh = open('myfile.txt', 'r')

for line in fh:
    if is_comment(line):
       use_comment(line)
    else:
       information = read_data(line)
```

```
for line in open('myfile.txt', 'r'):
    if is_comment(line):
        use_comment(line)
    else:
        information = read_data(line)
```

Program flow - while loops

```
keep_going = True
information = []
index = 0

while keep_going:
    current_line = lines[index]
    information += read_line(current_line)
    index += 1
    if check_something(current_line):
        keep_going = False
```

```
while keep_going:
    current_line = lines[index]
    information += read_line(current_line)
    index += 1
    if check_someting(current_line):
        keep_going = False
```

Different types of loops

`For` loop

is a control flow statement that performs operations over a known amount of steps.

`While` loop

is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition.

Which one to use?

`For` loops - standard for iterations over lists and other iterable objects

`While` loops - more flexible and can iterate an unspecified number of times

```python
user_input = "thank god it's friday"
for letter in user_input:
    print(letter.upper())
```

```
T
H
A
N
K

G
O
D

I
T
'
S

F
R
I
D
A
Y
```

In [57]:

```
i = 0
while i < len(user_input):
    letter = user_input[i]
    print(letter.upper())
    i += 1
```

```
T
H
A
N
K

G
O
D

I
T
'
S

F
R
I
D
A
Y
```

Controlling loops

- `break` - stop the loop
- `continue` - go on to the next iteration

```python
user_input = "thank god it's friday"
for letter in user_input:

    if letter == 'd':
        break
    print(letter.upper())
```

```
T
H
A
N
K

G
O
```

Watch out!

In [ ]:

```
# DON'T RUN THIS
i = 0
-while i < 10:
    print(user_input[i])
```

Watch out!

In [ ]:

```
# DON'T RUN THIS
i = 0
-while i < 10:
    print(user_input[i])
```

While loops may be infinite!

Input/Output

- In:
  - Read files: `fh = open(filename, 'r')`
    - `for line in fh:`
      - `fh.read()`
      - `fh.readlines()`
  - Read information from command line: `sys.argv[1:]`
- Out:
  - Write files: `fh = open(filename, 'w')`
    - `fh.write(text)`
  - Printing: `print('my_information')`

Input/Output

- Open files should be closed:
    - `fh.close()`

Code structure

- Functions
- Modules

Functions

- A named piece of code that performs a certain task.

```
def functionName(arg1, arg2, arg3):

    finalValue = 0

    # Here is some code where you can do
    # calculations etc, on arg1, arg2, arg3
    # and update finalValue

    return FinalValue
```

- Is given a number of input arguments
    - to be used (are in scope) within the function body
- Returns a result (maybe `None`)

Functions - keyword arguments

```
def prettyprinter(name, value, delim=":", end=None):
    out = "The " + name + " is " + delim + " " + value
    if end:
        out += end
    return out
```

- used to set default values (often `None`)
- can be skipped in function calls
- improve readability

Using your code

Any longer pieces of code that have been used and will be re-used should be saved

- Save it as a file `.py`
- To run it: `python3 mycode.py` or `python mycode.py`
- Import it: `import mycode`

Documentation and comments

- ```
  """ This is a doc-string explaining what the purpose of this
  function/module is """
  ```

- ```
  # This is a comment that helps understanding the code
  ```

Documentation and comments

- ```
  """ This is a doc-string explaining what the purpose of this
  function/module is """
  ```

- ```
  # This is a comment that helps understanding the code
  ```

- Comments *will* help you

Documentation and comments

- ```
  """ This is a doc-string explaining what the purpose of this
  function/module is """
  ```
- ```
  # This is a comment that helps understanding the code
  ```
- Comments *will* help you
- Undocumented code rarely gets used

Documentation and comments

- ```
  """ This is a doc-string explaining what the purpose of this
  function/module is """
  ```
- ```
  # This is a comment that helps understanding the code
  ```
- Comments *will* help you
- Undocumented code rarely gets used
- Try to keep your code readable: use informative variable and function names

Why programming?

Endless possibilities!

- reverse complement DNA
- custom filtering of VCF files
- plotting of results
- all excel stuff!

Why programming?

- Computers are fast
- Computers don't get bored
- Computers don't get sloppy

Why programming?

- Computers are fast
- Computers don't get bored
- Computers don't get sloppy
- Create reproducable results
    - for you and for others to use
- Extract large amount of information

Final advice

- Take a moment to think before you start coding
    - use pseudocode
    - use top-down programming
    - use paper and pen
    - take breaks

Final advice

- Take a moment to think before you start coding
    - use pseudocode
    - use top-down programming
    - use paper and pen
    - take breaks
- You know the basics - don't be afraid to try, it's the only way to learn
- You will get faster

Final advice (for real)

- Getting help
    - ask colleauges
    - try talk about your problem (get a rubber duck https://en.wikipedia.org/wiki/Rubber_duck_debugging)
    - search the web
    - NBIS drop-ins

Now you know Python!

Well done!

Just a small quiz to finish the day