

# Introduction to



## with Application to Bioinformatics

- Day 4

# TODAY

- Keyword arguments
- Loops with break and continue
- "Code structure": comments and documentation
- Importing modules: using libraries
- Pandas - explore your data!

## Review

- In what ways does the type of an object matter? Explain the output of:

In [2]:

```
row = 'sofa|2000|buy|Uppsala'
fields = row.split('|')
price = fields[1]
if price == 2000:
    print('The price is a number!')
if price == '2000':
    print('The price is a string!')
```

The price is a string!

In [3]:

```
print(sorted([ 2000, 30, 100 ]))
print(sorted(['2000', '30', '100']))
# Hint: is `30` > `2000`?
```

[30, 100, 2000]

['100', '2000', '30']

- How can you convert an object to a different type?
  - Convert to number: '2000' and '0.5' and '1e9'
  - Convert to boolean: 1, 0, '1', '0', '', {}
- We have seen these container types: **lists**, **sets**, **dictionaries**. What is their difference and when should you use which?
- What is a function? Write a function that counts the number of occurrences of 'C' in the argument string.

## In what ways does the type of an object matter?

```
In [4]: row = 'sofa|2000|buy|Uppsala'
fields = row.split('|')
price = int(fields[1])
if price == 2000:
    print('The price is a number!')
if price == '2000':
    print('The price is a string!')
```

The price is a number!

```
In [5]: print(sorted([ 2000, 30, 100 ]))
print(sorted(['2000', '30', '100']))
# Hint: is `30` > `2000`?
```

```
[30, 100, 2000]
['100', '2000', '30']
```

## In what ways does the type of an object matter?

- Each type store a specific type of information
  - `int` for integers,
  - `float` for floating point values (decimals),
  - `str` for strings,
  - `list` for lists,
  - `dict` for dictionaries.
- Each type supports different operations, functions and methods.

- Each type supports different **operations**, functions and methods

```
In [6]: 30 > 2000
```

```
Out[6]: False
```

```
In [7]: '30' > '2000'
```

```
Out[7]: True
```

```
In [8]: 30 > int('2000')
```

```
Out[8]: False
```

- Each type supports different operations, functions and **methods**

```
In [9]: 'ACTG'.lower()
```

```
Out[9]: 'actg'
```

```
In [10]: [1, 2, 3].lower()
```

```
-----  
AttributeError                                Traceback (most recent call last)  
<ipython-input-10-4e1a84c0439c> in <module>  
----> 1 [1, 2, 3].lower()
```

```
AttributeError: 'list' object has no attribute 'lower'
```

- Convert to number: '2000' and '0.5' and '1e9'

```
In [11]: int('2000')
```

```
Out[11]: 2000
```

```
In [12]: int('0.5')
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-12-6d0b04c882d1> in <module>  
----> 1 int('0.5')
```

```
ValueError: invalid literal for int() with base 10: '0.5'
```

```
In [13]: int('1e9')
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-13-cb568d180cc9> in <module>  
----> 1 int('1e9')
```

```
ValueError: invalid literal for int() with base 10: '1e9'
```

```
In [14]: float('2000')
```

```
Out[14]: 2000.0
```

```
In [15]: int(float('1.5'))
```

```
Out[15]: 1
```



In [16]: `int(float('1e9'))`

Out[16]: 1000000000

- Convert to boolean: 1, 0, '1', '0', '', {}

In [17]: `bool(1)`

Out[17]: True

In [18]: `bool(0)`

Out[18]: False

In [19]: `bool('1')`

Out[19]: True

In [20]: `bool('0')`

Out[20]: True

In [21]: `bool('')`

Out[21]: False

In [22]: `bool({})`

Out[22]: False

- Python and the truth: true and false values

In [23]:

```
values = [1, 0, '', '0', '1', [], [0]]
for x in values:
    if x:
        print(repr(x), 'is true!')
    else:
        print(repr(x), 'is false!')
```

```
1 is true!
0 is false!
'' is false!
'0' is true!
'1' is true!
[] is false!
[0] is true!
```

- Converting between strings and lists

In [24]: `list("hello")`

Out[24]: `['h', 'e', 'l', 'l', 'o']`

In [25]: `str(['h', 'e', 'l', 'l', 'o'])`

Out[25]: `"['h', 'e', 'l', 'l', 'o']"`

In [26]: `'_'.join(['h', 'e', 'l', 'l', 'o'])`

Out[26]: `'h_e_l_l_o'`

## Container types, when should you use which?

- **lists**: when order is important
- **dictionaries**: to keep track of the relation between keys and values
- **sets**: to check for membership. No order, no duplicates.

```
In [27]: genre_list = ["comedy", "drama", "drama", "sci-fi"]  
genre_list
```

```
Out[27]: ['comedy', 'drama', 'drama', 'sci-fi']
```

```
In [28]: genres = set(genre_list)  
'drama' in genres
```

```
Out[28]: True
```

```
In [29]: genre_counts = {"comedy": 1, "drama": 2, "sci-fi": 1}  
genre_counts
```

```
Out[29]: {'comedy': 1, 'drama': 2, 'sci-fi': 1}
```

```
In [30]: movie = {"rating": 10.0, "title": "Toy Story"}  
movie
```

```
Out[30]: {'rating': 10.0, 'title': 'Toy Story'}
```

## **What is a function?**

- A named piece of code that performs a specific task
- A relation (mapping) between inputs (arguments) and output (return value)
- Write a function that counts the number of occurrences of 'C' in the argument string.

- Function for counting the number of occurrences of 'C'

In [31]:

```
def cytosine_count(nucleotides):  
    count = 0  
    for x in nucleotides:  
        if x == 'c' or x == 'C':  
            count += 1  
    return count  
  
count1 = cytosine_count('CATATTAC')  
count2 = cytosine_count('tagtag')  
print(count1, count2)
```

2 0

- Functions that return are easier to repurpose than those that print their result

```
In [32]: cytosine_count('catattac') + cytosine_count('tactactac')
```

```
Out[32]: 5
```

```
In [33]: def print_cytosine_count(nucleotides):
          count = 0
          for x in nucleotides:
              if x == 'c' or x == 'C':
                  count += 1
          print(count)

          print_cytosine_count('catattac') + print_cytosine_count('tactactac')
```

```
2
3
```

---

```
TypeError                                 Traceback (most recent call last)
<ipython-input-33-5bbd47c30b94> in <module>
      6     print(count)
      7
----> 8 print_cytosine_count('catattac') + print_cytosine_count('tactactac')
```

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```



- Objects and references to objects

```
In [34]: list_A = ['red', 'green']  
list_B = ['red', 'green']  
list_B.append('blue')  
print(list_A, list_B)
```

```
['red', 'green'] ['red', 'green', 'blue']
```

```
In [35]: list_A = ['red', 'green']  
list_B = list_A      # another name to the SAME list. Aliasing  
list_B.append('blue')  
print(list_A, list_B)
```

```
['red', 'green', 'blue'] ['red', 'green', 'blue']
```

```
In [36]: list_A = ['red', 'green']  
list_B = list_A  
list_A = []  
print(list_A, list_B)
```

```
[] ['red', 'green']
```

- Objects and references to objects, cont.

In [37]:

```
list_A = ['red', 'green']
lists = {'A': list_A, 'B': list_A}
print(lists)
lists['B'].append('blue')
print(lists)
```

```
{'A': ['red', 'green'], 'B': ['red', 'green']}
{'A': ['red', 'green', 'blue'], 'B': ['red', 'green', 'blue']}
```

In [38]:

```
list_A = ['red', 'green']
lists = {'A': list_A, 'B': list_A}
print(lists)
lists['B'] = lists['B'] + ['yellow']
print(lists)
```

```
{'A': ['red', 'green'], 'B': ['red', 'green']}
{'A': ['red', 'green'], 'B': ['red', 'green', 'yellow']}
```

## Scope: global variables and local function variables

In [39]: `movies = ['Toy story', 'Home alone']`

In [40]: `def some_thriller_movies():  
 return ['Fargo', 'The Usual Suspects']  
  
movies = some_thriller_movies()  
print(movies)`

`['Fargo', 'The Usual Suspects']`

In [41]: `def change_to_drama(movies):  
 movies = ['Forrest Gump', 'Titanic']  
  
change_to_drama(movies)  
print(movies)`

`['Fargo', 'The Usual Suspects']`

In [42]: `def change_to_scifi(movies):  
 movies.clear()  
 movies += ['Terminator II', 'The Matrix']  
  
change_to_scifi(movies)  
print(movies)`

`['Terminator II', 'The Matrix']`

## Keyword arguments

- A way to give a name explicitly to a function for clarity

```
In [43]: sorted(list('file'), reverse=True)
```

```
Out[43]: ['l', 'i', 'f', 'e']
```

```
In [44]: attribute = 'gene_id "unknown gene"'
attribute.split(sep=' ', maxsplit=1)
```

```
Out[44]: ['gene_id', '"unknown gene"']
```

```
In [45]: # print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
print('x=', end='')
print('1')
```

```
x=1
```

## Keyword arguments

- Order of keyword arguments do not matter

```
open(file, mode='r', encoding=None) # some arguments omitted
```

- These mean the same:

```
open('files/recipes.txt', 'w', encoding='utf-8')
```

```
open('files/recipes.txt', mode='w', encoding='utf-8')
```

```
open('files/recipes.txt', encoding='utf-8', mode='w')
```

## Defining functions taking keyword arguments

- Just define them as usual:

```
In [46]: def format_sentence(subject, value, end):
          return 'The ' + subject + ' is ' + value + end

          print(format_sentence('lecture', 'ongoing', '.'))

          print(format_sentence('lecture', 'ongoing', end='.'))

          print(format_sentence(subject='lecture', value='ongoing', end='...'))
```

```
The lecture is ongoing.
The lecture is ongoing.
The lecture is ongoing...
```

```
In [47]: print(format_sentence(subject='lecture', 'ongoing', '.'))
```

```
File "<ipython-input-47-8916632389ec>", line 1
    print(format_sentence(subject='lecture', 'ongoing', '.'))
                                ^
```

**SyntaxError:** positional argument follows keyword argument

- Positional arguments comes first, keyword arguments after!

## Defining functions with default arguments

In [48]:

```
def format_sentence(subject, value, end='.'):
    return 'The ' + subject + ' is ' + value + end

print(format_sentence('lecture', 'ongoing'))

print(format_sentence('lecture', 'ongoing', '...'))
```

The lecture is ongoing.  
The lecture is ongoing...

## Defining functions with optional arguments

- Convention: use the object None

In [49]:

```
def format_sentence(subject, value, end='.', second_value=None):
    if second_value is None:
        return 'The ' + subject + ' is ' + value + end
    else:
        return 'The ' + subject + ' is ' + value + ' and ' + second_value + end

print(format_sentence('lecture', 'ongoing'))

print(format_sentence('lecture', 'ongoing',
                    second_value='self-referential', end='!'))
```

The lecture is ongoing.

The lecture is ongoing and self-referential!



## Small detour: Python's value for missing values: None

- Default value for optional arguments
- Implicit return value of functions without a return
- Something to initialize variable with no value yet
- Argument to a function indicating use the default value

```
In [50]: bool(None)
```

```
Out[50]: False
```

```
In [51]: None == False, None == 0
```

```
Out[51]: (False, False)
```

## Comparing None

- To differentiate None to the other false values such as 0, False and '' use `is None`:

```
In [52]: counts = {'drama': 2, 'romance': 0}
          counts.get('romance'), counts.get('thriller')
```

```
Out[52]: (0, None)
```

```
In [53]: counts.get('romance') is None
```

```
Out[53]: False
```

```
In [54]: counts.get('thriller') is None
```

```
Out[54]: True
```

- Python and the truth, take two

In [55]:

```
values = [None, 1, 0, '', '0', '1', [], [0]]
for x in values:
    if x is None:
        print(repr(x), 'is None')
    if not x:
        print(repr(x), 'is false')
    if x:
        print(repr(x), 'is true')
```

```
None is None
None is false
1 is true
0 is false
'' is false
'0' is true
'1' is true
[] is false
[0] is true
```

## Controlling loops - break


```
for x in lines_in_a_big_file:  
    if x.startswith('>'): # this is the only line I want!  
        do_something(x)
```

...waste of time!

```
for x in lines_in_a_big_file:  
    if x.startswith('>'): # this is the only line I want!  
        do_something(x)  
        break # break the loop
```

## break

```
for line in file:  
    if line.startswith('#'):  
        break  
    do_something(line)  
  
print("I am done")
```



## Controlling loops - continue


```
for x in lines_in_a_big_file:  
    if x.startswith('>'): # irrelevant line  
        # just skip this! don't do anything  
        do_something(x)
```

```
for x in lines_in_a_big_file:  
    if x.startswith('>'): # irrelevant line  
        continue # go on to the next iteration  
    do_something(x)
```

```
for x in lines_in_a_big_file:  
    if not x.startswith('>'): # not irrelevant!  
        do_something(x)
```

## continue

```
for line in file:  
    if line.startswith('#'):  
        continue  
    do_something(line)  
  
print("I am done")
```



## Another control statement: pass - the placeholder

In [56]:

```
def a_function():  
    # I have not implemented this just yet
```

File "<ipython-input-56-a7f30ec71867>", line 2

```
    # I have not implemented this just yet
```

^

**SyntaxError:** unexpected EOF while parsing

In [57]:

```
def a_function():  
    # I have not implemented this just yet  
    pass  
a_function()
```



## **Exercise 1**

- Notebook Day\_4\_Exercise\_1 (~30 minutes)

## **A short note on code structure**

- functions
- modules (files)
- documentation

## **Why functions?**

- Cleaner code
- Better defined tasks in code
- Re-usability
- Better structure

## **Why modules?**

- Cleaner code
  - Better defined tasks in code
  - Re-usability
  - Better structure
- 
- Collect all related functions in one file
  - Import a module to use its functions
  - Only need to understand what the functions do, not how

## Example: sys

```
import sys  
sys.argv[1]
```

or

```
import pprint  
pprint.pprint(a_big_dictionary)
```

# Python standard modules

Check out the [module index \(https://docs.python.org/3.6/py-modindex.html\)](https://docs.python.org/3.6/py-modindex.html).

How to find the right module?

How to understand it?

How to find the right module?

- look at the module index
- search PyPI (<http://pypi.org>)
- ask your colleagues
- search the web!



How to understand it?

In [58]:

```
import math  
help(math.acosh)
```

Help on built-in function acosh in module math:

`acosh(x, /)`

Return the inverse hyperbolic cosine of x.

In [59]: `help(str)`

Help on class str in module builtins:

```
class str(object)
|   str(object='') -> str
|   str(bytes_or_buffer[, encoding[, errors]]) -> str
|
|   Create a new string object from the given object. If encoding or
|   errors is specified, then the object must expose a data buffer
|   that will be decoded using the given encoding and error handler.
|   Otherwise, returns the result of object.__str__() (if defined)
|   or repr(object).
|   encoding defaults to sys.getdefaultencoding().
|   errors defaults to 'strict'.
|
|   Methods defined here:
|
|   __add__(self, value, /)
|       Return self+value.
|
|   __contains__(self, key, /)
|       Return key in self.
|
|   __eq__(self, value, /)
|       Return self==value.
|
|   __format__(self, format_spec, /)
|       Return a formatted version of the string as described by format_spec.
|
|   __ge__(self, value, /)
|       Return self>=value.
|
|   __getattr__(self, name, /)
|       Return getattr(self, name).
|
|   __getitem__(self, key, /)
```

```
    Return self[key].

__getnewargs__(...)

__gt__(self, value, /)
    Return self>value.

__hash__(self, /)
    Return hash(self).

__iter__(self, /)
    Implement iter(self).

__le__(self, value, /)
    Return self<=value.

__len__(self, /)
    Return len(self).

__lt__(self, value, /)
    Return self<value.

__mod__(self, value, /)
    Return self%value.

__mul__(self, value, /)
    Return self*value.

__ne__(self, value, /)
    Return self!=value.

__repr__(self, /)
    Return repr(self).

__rmod__(self, value, /)
    Return value%self.

__rmul__(self, value, /)
```

Return value\*self.

`__sizeof__(self, /)`

Return the size of the string in memory, in bytes.

`__str__(self, /)`

Return `str(self)`.

`capitalize(self, /)`

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

`casefold(self, /)`

Return a version of the string suitable for caseless comparisons.

`center(self, width, fillchar=' ', /)`

Return a centered string of length `width`.

Padding is done using the specified fill character (default is a space).

`count(...)`

`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.

`encode(self, /, encoding='utf-8', errors='strict')`

Encode the string using the codec registered for encoding.

`encoding`

The encoding in which to encode the string.

`errors`

The error handling scheme to use for encoding errors.

The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' an

'xmlcharrefreplace' as well as any other name registered with codecs.register\_error that can handle UnicodeEncodeErrors.

endswith(...)

S.endswith(suffix[, start[, end]]) -> bool

Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs(self, /, tabsize=8)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find(...)

S.find(sub[, start[, end]]) -> int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(...)

S.format(\*args, \*\*kwargs) -> str

Return a formatted version of S, using substitutions from args and kwars.

The substitutions are identified by braces ('{' and '}').

format\_map(...)

S.format\_map(mapping) -> str

Return a formatted version of S, using substitutions from mapping.  
The substitutions are identified by braces ('{' and '}').

`index(...)`

`S.index(sub[, start[, end]]) -> int`

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

`isalnum(self, /)`

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

`isalpha(self, /)`

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

`isascii(self, /)`

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F.  
Empty string is ASCII too.

`isdecimal(self, /)`

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

`isdigit(self, /)`

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

`isidentifier(self, /)`

Return True if the string is a valid Python identifier, False otherwise.

Call `keyword.iskeyword(s)` to test whether string `s` is a reserved identifier, such as "def" or "class".

`islower(self, /)`

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

`isnumeric(self, /)`

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

`isprintable(self, /)`

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in `repr()` or if it is empty.

`isspace(self, /)`

Return True if the string is a whitespace string, False otherwise.

nd there  
A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle(self, /)

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper(self, /)

Return True if the string is an uppercase string, False otherwise.

se and  
A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(self, iterable, /)

Concatenate any number of strings.

ing.  
The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

ljust(self, width, fillchar=' ', /)

Return a left-justified string of length width.

e).  
Padding is done using the specified fill character (default is a space).

lower(self, /)

Return a copy of the string converted to lowercase.

rstrip(self, chars=None, /)



Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

partition(self, sep, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

replace(self, old, new, count=-1, /)

Return a copy with all occurrences of substring old replaced by new.

count

Maximum number of occurrences to replace.

-1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(...)

S.rfind(sub[, start[, end]]) -> int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(...)

S.rindex(sub[, start[, end]]) -> int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust(self, width, fillchar=' ', /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(self, sep, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end.

If  
e

the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit(self, /, sep=None, maxsplit=-1)

Return a list of the words in the string, using sep as the delimiter string.

sep

The delimiter according which to split the string.

None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit

Maximum number of splits to do.

-1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

`rstrip(self, chars=None, /)`

Return a copy of the string with trailing whitespace removed.

If `chars` is given and not `None`, remove characters in `chars` instead.

`split(self, /, sep=None, maxsplit=-1)`

Return a list of the words in the string, using `sep` as the delimiter string.

`sep`

The delimiter according which to split the string.

`None` (the default value) means split according to any whitespace, and discard empty strings from the result.

`maxsplit`

Maximum number of splits to do.

`-1` (the default value) means no limit.

`splitlines(self, /, keepends=False)`

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless `keepends` is given and `true`.

`startswith(...)`

`S.startswith(prefix[, start[, end]]) -> bool`

Return `True` if `S` starts with the specified prefix, `False` otherwise.

With optional `start`, test `S` beginning at that position.

With optional `end`, stop comparing `S` at that position.

`prefix` can also be a tuple of strings to try.

`strip(self, chars=None, /)`

Return a copy of the string with leading and trailing whitespace removed.

d.

If `chars` is given and not `None`, remove characters in `chars` instead.

`swapcase(self, /)`

Convert uppercase characters to lowercase and lowercase characters to uppercase.

`title(self, /)`

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining  
cased characters have lower case.

`translate(self, table, /)`

Replace each character in the string using the given translation table.

`table`

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or `None`.

The table must implement lookup/indexing via `__getitem__`, for instance  
a  
dictionary or list. If this operation raises `LookupError`, the character  
r is  
left untouched. Characters mapped to `None` are deleted.

`upper(self, /)`

Return a copy of the string converted to uppercase.

`zfill(self, width, /)`

Pad a numeric string with zeros on the left, to fill a field of the given  
width.

The string is never truncated.

---

Static methods defined here:

`__new__(*args, **kwargs)` from `builtins.type`

Create and return a new object. See `help(type)` for accurate signature.

`maketrans(...)`

Return a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals.

If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in `x` will be mapped to the character at the same position in `y`. If there is a third argument, it must be a string, whose characters will be mapped to None in the resul

t.

In [60]:

```
help(math.sqrt)
```

```
# install packages using: pip
```

```
# Dimitris' protip: install packages using conda
```

Help on built-in function sqrt in module math:

```
sqrt(x, /)
```

```
    Return the square root of x.
```

In [61]:

```
math.sqrt(3)
```

Out[61]: 1.7320508075688772

## Importing

```
In [62]: import math  
math.sqrt(3)
```

Out[62]: 1.7320508075688772

```
In [63]: import math as m  
m.sqrt(3)
```

Out[63]: 1.7320508075688772

```
In [64]: from pprint import pprint
```



# Documentation and commenting your code

Remember `help()` ?

Works because somebody else has documented their code!

In [65]:

```
def process_file(filename, chrom, pos):  
    """  
    Read a vcf file, search for lines matching  
    chromosome chrom and position pos.  
  
    Print the genotypes of the matching lines.  
    """  
    for line in open(filename):  
        if not line.startswith('#'):  
            col = line.split('\t')  
            if col[0] == chrom and col[1] == pos:  
                print(col[9:])  
help(process_file)
```

Help on function process\_file in module `__main__`:

```
process_file(filename, chrom, pos)  
    Read a vcf file, search for lines matching  
    chromosome chrom and position pos.  
  
    Print the genotypes of the matching lines.
```

In [66]:

```
help(process_file)
```

Help on function process\_file in module \_\_main\_\_:

```
process_file(filename, chrom, pos)
```

Read a vcf file, search for lines matching chromosome chrom and position pos.

Print the genotypes of the matching lines.

Your code may have two types of users:

- library users
- maintainers (maybe yourself!)

Write documentation for both of them!

- library users (docstrings):

```
"""  
What does this function do?  
"""
```

- maintainers (comments):

```
# implementation details
```

## Documentation:

- At the beginning of the file

```
"""  
    This module provides functions fo  
    r...  
    """  
□
```

- For every function

```
def make_list(x):  
    """Returns a random list of length  
    x."""  
    pass
```

## Comments:

- Wherever the code is hard to understand

```
my_list[5] += other_list[3] # explain why you do this!
```

## Read more:

<https://realpython.com/documenting-python-code/> (<https://realpython.com/documenting-python-code/>)

<https://www.python.org/dev/peps/pep-0008/?#comments>  
(<https://www.python.org/dev/peps/pep-0008/?#comments>)

# Formatting

```
In [68]: title = 'Toy Story'
         rating = 10
         print('The result is: ' + title + ' with rating: ' + str(rating))
```

The result is: Toy Story with rating: 10

```
In [69]: # f-strings (since python 3.6)
         print(f'The result is: {title} with rating: {rating}')
```

The result is: Toy Story with rating: 10

```
In [70]: # format method
         print('The result is: {} with rating: {}'.format(title, rating))
```

The result is: Toy Story with rating: 10

```
In [71]: # the ancient way (python 2)
         print('The result is: %s with rating: %s' % (title, rating))
```

The result is: Toy Story with rating: 10

Learn more from the Python docs: <https://docs.python.org/3.4/library/string.html#format-string-syntax> (<https://docs.python.org/3.4/library/string.html#format-string-syntax>)

## Exercise 2

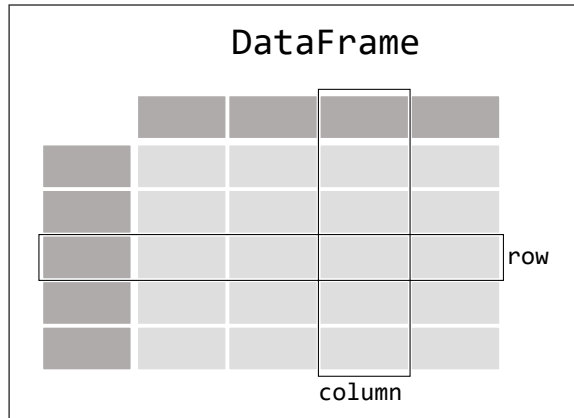
```
pick_movie(year=1996, rating_min=8.5)
The Bandit
pick_movie(rating_max=8.0, genre="Mystery")
Twelve Monkeys
```

- Notebook Day\_4\_Exercise\_2



# Pandas

- Library for working with tabular data
- Data analysis:
  - filter
  - transform
  - aggregate
  - plot
- Main hero: the DataFrame type:



## Creating a small DataFrame

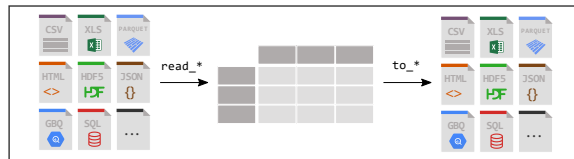
```
In [72]: import pandas as pd
df = pd.DataFrame({
    'age': [1,2,3,4],
    'circumference': [2,3,5,10],
    'height': [30, 35, 40, 50]
})
df
```

```
Out[72]:
```

	age	circumference	height
0	1	2	30
1	2	3	35
2	3	5	40
3	4	10	50

## Pandas can import data from many formats

- `pd.read_table`: tab separated values `.tsv`
- `pd.read_csv`: comma separated values `.csv`
- `pd.read_excel`: Excel spreadsheets `.xlsx`
- For a data frame `df`: `df.write_table()`, `df.write_csv()`, `df.write_excel()`



## Orange tree data

In [73]: `!cat ../downloads/Orange_1.tsv`

```
age      circumference  height
1         2           30
2         3           35
3         5           40
4        10           50
```

In [74]: `df = pd.read_table('../downloads/Orange_1.tsv')`  
`df`

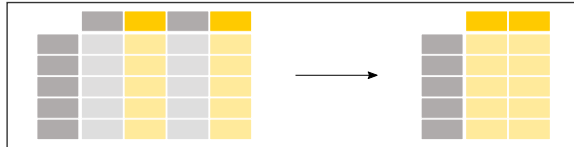
Out[74]:

	age	circumference	height
0	1	2	30
1	2	3	35
2	3	5	40
3	4	10	50

- One implicit index (0, 1, 2, 3)
- Columns: age, circumference, height
- Rows: one per data point, identified by their index

## Selecting columns from a dataframe

```
dataframe.columnname  
dataframe['columnname']
```



```
In [75]: df.columns
```

```
Out[75]: Index(['age', 'circumference', 'height'], dtype='object')
```

```
In [76]: df[['height', 'age']]
```

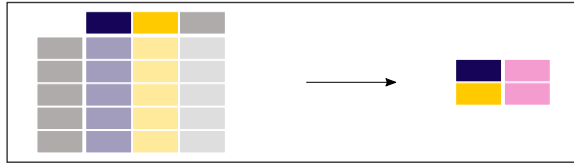
```
Out[76]:
```

	height	age
0	30	1
1	35	2
2	40	3
3	50	4

```
In [77]: df.height
```

```
Out[77]: 0    30  
         1    35  
         2    40  
         3    50  
         Name: height, dtype: int64
```

## Calculating aggregated summary statistics



```
In [78]: df[['age', 'circumference']].describe()
```

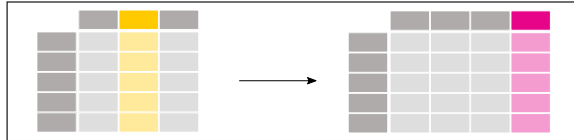
```
Out[78]:
```

	age	circumference
count	4.000000	4.000000
mean	2.500000	5.000000
std	1.290994	3.559026
min	1.000000	2.000000
25%	1.750000	2.750000
50%	2.500000	4.000000
75%	3.250000	6.250000
max	4.000000	10.000000

```
In [79]: df['age'].std()
```

```
Out[79]: 1.2909944487358056
```

## Creating new column derived from existing column



```
In [80]: import math
df['radius'] = df['circumference'] / 2.0 / math.pi
df
```

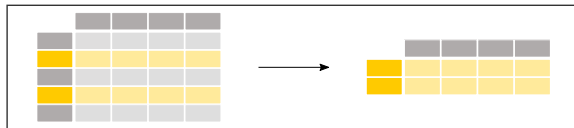
```
Out[80]:
```

	age	circumference	height	radius
0	1	2	30	0.318310
1	2	3	35	0.477465
2	3	5	40	0.795775
3	4	10	50	1.591549



## Selecting rows from a dataframe by index

```
dataframe.iloc[index]  
dataframe.iloc[start:stop]
```



```
In [81]: df.iloc[1:3]
```

```
Out[81]:
```

	age	circumference	height	radius
1	2	3	35	0.477465
2	3	5	40	0.795775

## Slightly bigger data frame of orange trees

```
In [82]: !head -n 6 ../downloads/Orange.tsv
```

```
Tree    age    circumference
1       118     30
1       484     58
1       664     87
1      1004    115
1      1231    120
```

```
In [83]: df = pd.read_table('../downloads/Orange.tsv') # , index_col=0)
df.iloc[0:5] # can also use .head()
```

```
Out[83]:
```

	Tree	age	circumference
0	1	118	30
1	1	484	58
2	1	664	87
3	1	1004	115
4	1	1231	120

```
In [84]: df.Tree.unique()
```

```
Out[84]: array([1, 2, 3])
```

```
In [85]: type(pd.DataFrame({"genre": ['Thriller', 'Drama'], "rating": [10, 9]}).rating.iloc[0])
```

```
Out[85]: numpy.int64
```

In [86]:

```
#young = df[df.age < 200]  
#young  
df[df.age < 1000]
```

Out[86]:

	Tree	age	circumference
0	1	118	30
1	1	484	58
2	1	664	87
7	2	118	33
8	2	484	69
9	2	664	111
14	3	118	30
15	3	484	51
16	3	664	75

## Finding the maximum and then filter by it

```
df.loc[ df.age < 200 ]
```

In [87]:

```
df.head()
```

Out[87]:

	Tree	age	circumference
0	1	118	30
1	1	484	58
2	1	664	87
3	1	1004	115
4	1	1231	120

In [88]:

```
max_c = df.circumference.max()  
print(max_c)
```

203

In [89]:

```
df[df.circumference == max_c]
```

Out[89]:

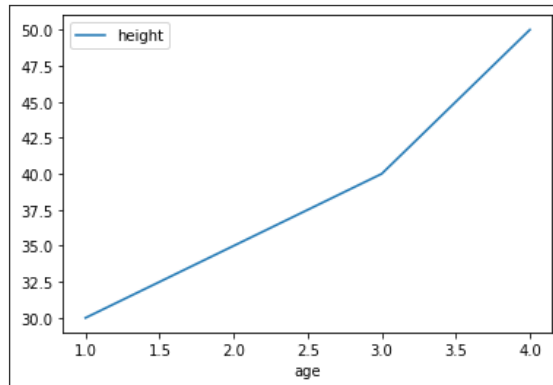
	Tree	age	circumference
12	2	1372	203
13	2	1582	203

# Plotting

```
df.columnname.plot()
```

```
In [90]: small_df = pd.read_table('../downloads/Orange_1.tsv')  
small_df.plot(x='age', y='height')
```

```
Out[90]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3f43b912e0>
```



## Plotting

What if no plot shows up?

```
%pylab inline # jupyter notebooks
```

or

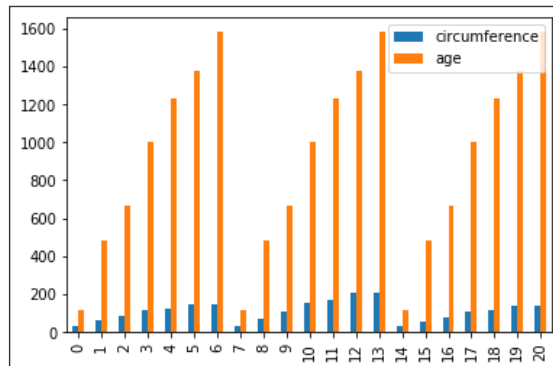
```
import matplotlib.pyplot as plt  
plt.show()
```

## Plotting - many trees

- Plot a bar chart

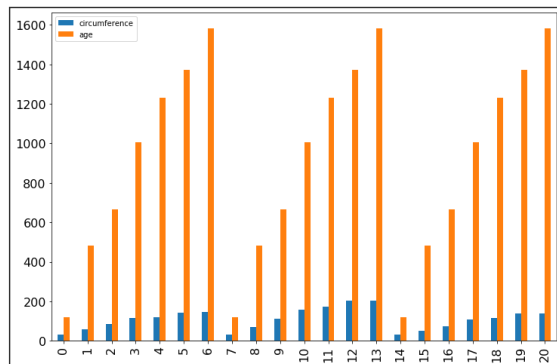
```
In [91]: df[['circumference', 'age']].plot(kind='bar')
```

```
Out[91]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3f4348c820>
```



```
In [92]: df[['circumference', 'age']].plot(kind='bar', figsize=(12, 8), fontsize=16)
```

```
Out[92]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3f433e1ee0>
```



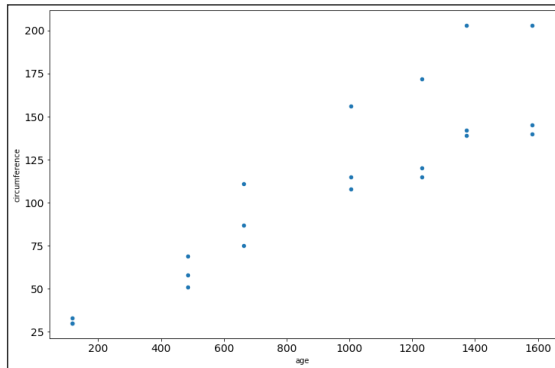


## Scatterplot

```
df.plot(kind="scatter", x="column_name", y="other_column_name")
```

```
In [93]: df.plot(kind='scatter', x='age', y='circumference',  
              figsize=(12, 8), fontsize=14)
```

```
Out[93]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3f43419a90>
```

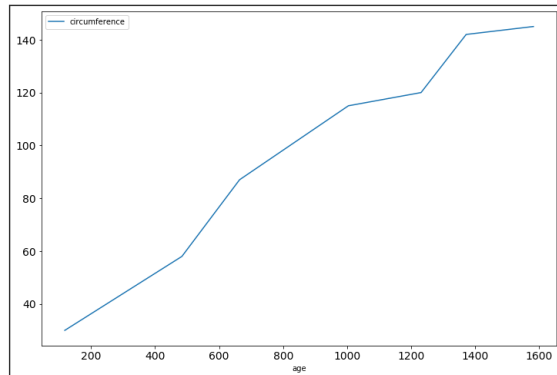


## Line plot

```
dataframe.plot(kind="line", x=..., y=...)
```

```
In [94]: tree1 = df[df['Tree'] == 1]
tree1.plot(x='age', y='circumference',
           fontsize=14, figsize=(12,8))
```

```
Out[94]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3f43295a00>
```



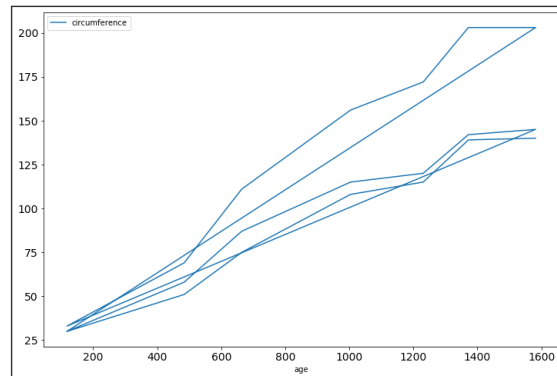
## Line plot of all trees

- Let's plot all the trees

```
dataframe.plot(kind="line", x="..", y=
"...")
```

```
In [95]: df.plot(kind='line', x='age', y='circumference',
               figsize=(12, 8), fontsize=14)
```

```
Out[95]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3f431f2c40>
```



:(

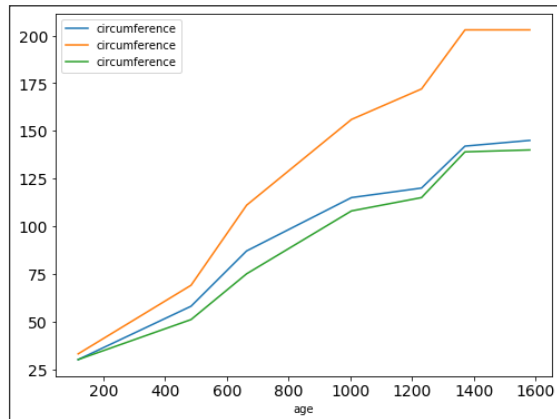
In [96]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

tree_names = df['Tree'].unique()
print('tree_names:', tree_names)

for tree_name in tree_names:
    sub_df = df[df['Tree'] == tree_name]
    sub_df.plot(x='age', y='circumference', kind='line',
               ax=ax, fontsize=14, figsize=(8,6))
```

tree\_names: [1 2 3]



## Exercise 5

- Read the `Orange_1.tsv`
  - Print the height column
  - Print the data for the tree at age 2
  - Find the maximum circumference
  - What tree reached that circumference, and how old was it at that time?
- Use Pandas to read IMDB
  - Explore it by making graphs
- Extra exercises:
  - Read the pandas documentation :)
  - Look at seaborn for a more feature-rich plotting lib