

Introduction to



with Application to Bioinformatics

- Day 4

Review

- Why does it matter what type a variable has?
 - What is the difference between `'1'*2` and `1*2`?
 - Is `'2' < '12'`?
- How (and when) can you change the type of those? When does it not work?
 - How do you make the string `'0.29'` into a number?
- You have worked with a number of data containers; lists, sets, dictionaries. What is the difference between them and when should you use which?
- What is a function?
 - Write a function that multiplies the input argument by 2.

Review

- Why does it matter what type a variable has?

Review

- Why does it matter what type a variable has?

Values of different types stores different types of information.

Different types can be used with different operations, functions and methods.

1+2

In []: "1"+"2"

In []: "1"+2

In []: "1"*2

Take extra care when comparing values

In []: 2 < 12

In []: "2" < "12"

Review

- How can you change (convert) the type of a value? When does it not work?

Review

- How can you change (convert) the type of a value? When does it not work?

In []: `float("1")`

In []: `int("1")`

In []: `str(1)`

Review

- How can you change (convert) the type of a value? When does it not work?

In []: `float("1")`

In []: `int("1")`

In []: `str(1)`

In []: `int("2.2")`

Converting between strings and lists

In []:

```
list("hello")
```

Converting between strings and lists

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

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

Converting between strings and lists

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

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

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

Review

- You have worked with a number of data containers; lists, sets, dictionaries. What is the difference between them and when should you use which?

Review

- You have worked with a number of data containers; lists, sets, dictionaries. What is the difference between them and 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.

Review

- You have worked with a number of data containers; lists, sets, dictionaries. What is the difference between them and 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 [ ]: mylist = ["comedy", "drama", "drama", "sci-fi"]  
        myset = set(mylist)  
        print('All genres', myset)  
        mydict = {"genre": "drama", "title": "Toy Story"}
```

Review

- What is a function?

Review

- What is a function?

A named piece of code that performs a specific task.

A relation between input data (arguments) and a result (output data).

Function structure

```
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
```

Function structure

```
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
```

- The def keyword

Function structure

```
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
```

- The def keyword
- Arguments

Function structure

```
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
```

- The def keyword
- Arguments
- Indentation!

Function structure

```
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
```

- The def keyword
- Arguments
- Indentation!
- return

Multiply by two

```
In [ ]: def multiply_by_two(x):  
        return x*2  
  
multiply_by_two(2)
```

```
In [ ]: def multiply_by_two(x):  
        print(x*2)  
  
multiply_by_two(2)
```

Multiply by two - return values

Multiply by two - return values

```
In [ ]: def multiply_by_two(x):  
        return x*2  
  
        result = multiply_by_two(2)  
        print('Result', result)
```

Multiply by two - return values

```
In [ ]: def multiply_by_two(x):  
        return x*2  
  
        result = multiply_by_two(2)  
        print('Result', result)
```

```
In [ ]: def multiply_by_two(x):  
        print(x*2)  
  
        result = multiply_by_two(2)  
        print('Result', result)
```

Multiply by two - return values

```
In [ ]: def multiply_by_two(x):  
        return x*2  
  
        result = multiply_by_two(2)  
        print('Result', result)
```

```
In [ ]: def multiply_by_two(x):  
        print(x*2)  
  
        result = multiply_by_two(2)  
        print('Result', result)
```

```
In [ ]: def multiply_by_two(x):  
        x*2  
  
        res = multiply_by_two(2)  
        print('Result', result)
```

TODAY

- Loops and functions, code structure
- Pandas - explore your data!

Common concepts

Scope

Scope

In []:

a

Scope

In []:

```
a
```

In []:

```
def myfunc(a, b):  
    print(a)
```

```
a
```


Scope

In []:

```
a
```

In []:

```
def myfunc(a, b):  
    print(a)  
  
a
```

In []:

```
a = 5  
  
def myfunc(a):  
    a += 2  
    return a  
  
b = myfunc(8)  
  
a, b
```

Scope

In []:

```
a
```

In []:

```
def myfunc(a, b):  
    print(a)  
  
a
```

In []:

```
a = 5  
  
def myfunc(a):  
    a += 2  
    return a  
  
b = myfunc(8)  
  
a, b
```

The local `a` in `myfunc` *shadows* the global `a`.

Scope

In []:

```
a
```

In []:

```
def myfunc(a, b):  
    print(a)  
  
a
```

In []:

```
a = 5  
  
def myfunc(a):  
    a += 2  
    return a  
  
b = myfunc(8)  
  
a, b
```

The local `a` in `myfunc` *shadows* the global `a`.

The variables inside functions are *local*. To avoid confusion, use different variable names.

In []:

```
global_a = 5

def myfunc(value):
    value += 2
    return value

result = myfunc(8)

global_a, result
```

No confusion!

myfunc has unique variable names.

Return values

```
In [ ]: def multiply_by_two(x):  
        return x*2  
  
        result = multiply_by_two(2)  
        print('Result', result)
```

```
In [ ]: def multiply_by_two(x):  
        print(x*2)  
  
        result = multiply_by_two(2)  
        print('Result', result)
```

```
In [ ]: def multiply_by_two(x):  
        x*2  
  
        res = multiply_by_two(2)  
        print('Result', result)
```

None

None

What is None? Why? When is it used?

None

What is None? Why? When is it used?

- What is it?
 - A keyword with a constant value (like True, False)
 - Null

None

What is None? Why? When is it used?

- What is it?
 - A keyword with a constant value (like `True`, `False`)
 - `Null`
- Why and when?
 - To signal "empty values"
 - Variables with no values yet
 - Functions that don't return anything meaningful

Comparing None

Comparing None

```
In [ ]: value = None
        if value is not None:
            print('value is something')
        else:
            print('no value!')
```

```
In [ ]: None == True
```

```
In [ ]: None == False
```

Comparing None

In []:

```
myvalue = None
if myvalue > 10:
    print('Big value')
```

Comparing None

```
In [ ]: myvalue = None  
        if myvalue > 10:  
            print('Big value')
```

```
In [ ]: myvalue = None  
        if myvalue is not None:  
            print('The value is not None')  
        else:  
            print('The value is None')
```

Exercise A

Write a function that takes one argument called `value`.

- If `value` is not `None` and if it is greater to or equal to 10, the function should print "big number".
- If it is not `None`, and if it is between 0 and 10, the function should print "small number".
- Otherwise, the function should print "No number".

```
In [ ]: def tester(value):  
        if value is not None and value >= 10:  
            print("big number")  
        elif value is not None and 0 < value < 10:  
            print("small number")  
        else:  
            print("No number")
```

```
In [ ]: tester(11)
```

```
In [ ]: tester(3)
```

```
In [ ]: tester(None)
```

Keyword arguments

Keyword arguments

```
open('../files/250.imdb', 'r', encoding='utf-8')
```

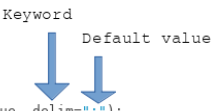
```
In [ ]: def prettyprinter(name, value, delim=":"):
        out = "The " + name + " is " + delim + " " + value + "."
        return out
```

```
In [ ]: def prettyprinter(name, value, delim=":"):
        out = "The " + name + " is " + delim + " " + value + "."
        return out
```

- Programmer can set default values

```
In [ ]: def prettyprinter(name, value, delim=":"):
        out = "The " + name + " is " + delim + " " + value + "."
        return out
```

- Programmer can set default values

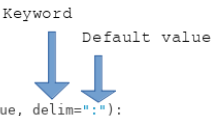


The diagram shows two blue arrows pointing down to the 'delim' parameter in the function signature. The left arrow is labeled 'Keyword' and points to the colon. The right arrow is labeled 'Default value' and points to the string ': '.

```
def prettyprinter(name, value, delim=":"):
    out = "The " + name + " is " + delim + " " + value + "."
    return out
```

```
In [ ]: def prettyprinter(name, value, delim=":"):
        out = "The " + name + " is " + delim + " " + value + "."
        return out
```

- Programmer can set default values



The diagram shows two blue arrows pointing down to the 'delim' parameter in the function definition. The left arrow is labeled 'Keyword' and points to the colon in 'delim=:'. The right arrow is labeled 'Default value' and points to the string ':"'.

```
def prettyprinter(name, value, delim=:):
    out = "The " + name + " is " + delim + " " + value + "."
    return out
```

```
In [ ]: prettyprinter("title", "Movie")
```

- User can ignore the arguments (default value is used)

```
def prettyprinter(name, value, delim=":"):
    out = "The " + name + " is " + delim + " " + value + "."
    return out
```

```
def prettyprinter(name, value, delim=":"):
    out = "The " + name + " is " + delim + " " + value + "."
    return out
```

```
In [ ]: prettyprinter("genre", "Drama", delim="=")
```

```
def prettyprinter(name, value, delim=":"):
    out = "The " + name + " is " + delim + " " + value + "."
    return out
```

```
In [ ]: prettyprinter("genre", "Drama", delim="=")
```

```
In [ ]: prettyprinter("genre", "Drama", "=")
```



```
In [ ]: def prettyprinter(name, value, delim=":", end=None):  
        out = "The " + name + " is " + delim + " " + value  
        if end:  
            out += end  
        return out  
  
        my_str = prettyprinter("title", "Movie")  
        print(my_str)
```

```
In [ ]: def prettyprinter(name, value, delim=":", end=None):  
        out = "The " + name + " is " + delim + " " + value  
        if end:  
            out += end  
        return out  
  
        my_str = prettyprinter("title", "Movie")  
        print(my_str)
```

```
In [ ]: prettyprinter("genre", "Drama", "=", ".")
```

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

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

- Gives better overview

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

- Gives better overview

```
open('../files/250.imdb', 'r', encoding='utf-8')
```

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

- Gives better overview

```
open('../files/250.imdb', 'r', encoding='utf-8')
```

```
open('../files/250.imdb', mode='r', encoding='utf-8')
```

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

- Gives better overview

```
open('../files/250.imdb', 'r', encoding='utf-8')
```

```
open('../files/250.imdb', mode='r', encoding='utf-8')
```

```
open('../files/250.imdb', encoding='utf-8', mode='r')
```

Keyword arguments

- programmer: set default values
- user: ignore arguments
- better overview

Exercise B

Rewrite the function from previous exercise, so that it takes one **keyword argument**, `value`. The default value of this argument is `None`.

- If `value` is not `None` and if it is greater to or equal to 10, the function should print "big number".
- If it is not `None`, and if it is between 0 and 10, the function should print "small number".
- Otherwise, the function should print "No number".

```
In [ ]: def tester(value=None):  
        if value is not None and value >= 10:  
            print("big number")  
        elif value is not None and 0 < value < 10:  
            print("small number")  
        else:  
            print("No number")
```

```
In [ ]: tester(11)
```

```
In [ ]: tester(3)
```

```
In [ ]: tester()
```

Controlling loops - break

Controlling loops - **break**

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

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!

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

Controlling loops - `continue`

```
for x in lines_in_a_big_file:
    if x.startswith('>'): # this is a comment
        # just skip this! don't do anything
        continue
    do_something(x)
```

Controlling loops - **continue**

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

```
for x in lines_in_a_big_file:
    if x.startswith('>'): # this is a comment
        continue # go on to the next iteration
    do_something(x)
```

Controlling loops - `continue`

```
for x in lines_in_a_big_file:
    if x.startswith('>'): # this is a comment
        # just skip this! don't do anything
        continue
    do_something(x)
```


```
for x in lines_in_a_big_file:
    if x.startswith('>'): # this is a comment
        continue # go on to the next iteration
    do_something(x)
```

```
for x in lines_in_a_big_file:
    if not x.startswith('>'): # this is *not* a comment
        do_something(x)
```

continue

```
for line in file:
    if line.startswith('#'):
        continue
    do_something(line)

print("I am done")
```



Exercise 1

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

→ **Notebook Day_4_Exercise_1**

A short note on code structure

- functions
- modules (files)
- documentation

Remember?

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

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 imbd_parser as imdb  
imdb.parse('250.imdb')
```

Python standard modules

Check out the module index (<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?

How to understand it?

```
In [ ]: import math  
        help(math)
```

In []:

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


In []:

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

Importing

In []:

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

Importing

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

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

Importing

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

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

```
In [ ]: from math import sqrt  
        sqrt(3)
```

Documentation and commenting your code

Remember `help()`?

Works because somebody else has documented their code!

```
In [ ]: def process_file(filename, chrom, pos):  
        for line in open(filename):  
            if not line.startswith('#'):  
                columns = line.split('\t')  
                if col[0] == chrom and col[1] == pos:  
                    print(col[9:])
```

```
In [ ]: def process_file(filename, chrom, pos):  
        for line in open(filename):  
            if not line.startswith('#'):  
                columns = line.split('\t')  
                if col[0] == chrom and col[1] == pos:  
                    print(col[9:])
```

?

```
In [ ]: def process_file(filename, chrom, pos):
        for line in open(filename):
            if not line.startswith('#'):
                columns = line.split('\t')
                if col[0] == chrom and col[1] == pos:
                    print(col[9:])
```

?

```
In [ ]: 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('#'):
                columns = line.split('\t')
                if col[0] == chrom and col[1] == pos:
                    print(col[9:])
```



```
In [ ]: def process_file(filename, chrom, pos):
        for line in open(filename):
            if not line.startswith('#'):
                columns = line.split('\t')
                if col[0] == chrom and col[1] == pos:
                    print(col[9:])
```

?

```
In [ ]: 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('#'):
                columns = line.split('\t')
                if col[0] == chrom and col[1] == pos:
                    print(col[9:])
```

```
In [ ]: help(process_file)
```

Your code may have two types of users:

- library users
- maintainers (maybe yourself!)

Your code may have two types of users:

- library users
- maintainers (maybe yourself!)

Write documentation for both of them!

- library users: `"""What does this function do?"""` (doc strings)
- maintainers: `# implementation details` (comments)

Documentation:

Documentation:

- At the beginning of the file

```
"""This module provides functions for..."""
```

Comments:

Comments:

```
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

Formatting

```
In [ ]: title = 'Great movie'  
        rating = 10  
        print('The result is: ' + title + 'with rating: ' + rating)
```

Formatting

```
In [ ]: title = 'Great movie'  
        rating = 10  
        print('The result is: ' + title + 'with rating: ' + rating)
```

Formatting!

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

Formatting

```
In [ ]: title = 'Great movie'
        rating = 10
        print('The result is: ' + title + 'with rating: ' + rating)
```

Formatting!

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

Formatting - f-strings

```
In [ ]: print(f'The result is: {title} with rating {rating}') # python version >= 3.6
```

Formatting

```
In [ ]: title = 'Great movie'
        rating = 10
        print('The result is: ' + title + 'with rating: ' + rating)
```

Formatting!

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

Formatting - f-strings

```
In [ ]: print(f'The result is: {title} with rating {rating}') # python version >= 3.6
```

Formatting - the old way

```
In [ ]: print('The result is: %s with rating %s' % (title, rating)) # python2
```

Formatting

Formatting

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

→ Notebook Day_4_Exercise_2

Pandas

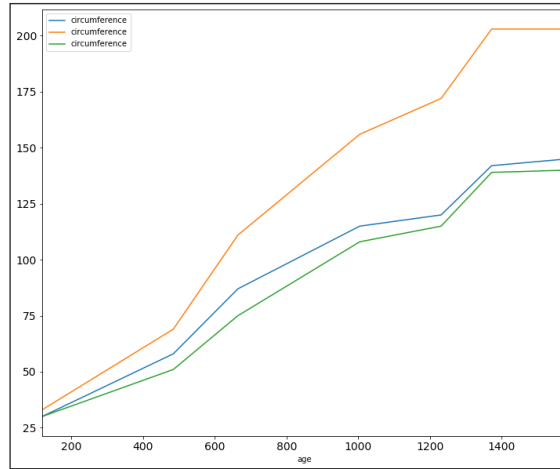
Library for working with tabular data

- comma separated (csv)
- tab separated (tsv)
- ...

Data analysis, graph plotting...

Pandas

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



Pandas - a short overview

```
In [ ]: import pandas as pd
```

```
In [ ]: help(pd)
```

Orange tree data

Orange_1.tsv:

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

Orange tree data

Orange_1.tsv:

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

```
In [ ]: tree_growth = pd.read_table('../downloads/Orange_1.tsv', index_col=0)
```

Dataframes

In []: `tree_growth`

- One index (in this case age)
- A bunch of columns (in this case `circumference` and `height`)
- A bunch of rows (identified by their index)

Dataframes

In []: `tree_growth`

- One index (in this case `age`)
- A bunch of columns (in this case `circumference` and `height`)
- A bunch of rows (identified by their index)

In []: `tree_growth.columns`

Dataframes

In []: `tree_growth`

- One index (in this case `age`)
- A bunch of columns (in this case `circumference` and `height`)
- A bunch of rows (identified by their index)

In []: `tree_growth.columns`

In []: `tree_growth.index`

Exploring the data - picking a column

Exploring the data - picking a column

In []: `tree_growth.circumference`

`dataframe.columnname`

`dataframe['columnname']`

Exploring the data - picking a column

```
In [ ]: tree_growth.circumference
```

```
dataframe.columnname
```

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

```
In [ ]: tree_growth.height
```

Exploring the data - picking a column

```
In [ ]: tree_growth.circumference
```

```
dataframe.columnname
```

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

```
In [ ]: tree_growth.height
```

```
In [ ]: tree_growth.circumference.max()
```

Exploring the data - picking a row

Exploring the data - picking a row

In []: `tree_growth.loc[4]`

`dataframe.loc[row_name]`

Reading data

```
dataframe = pandas.read_table(filepath, index_col=N)
```

```
dataframe.columnname
```

```
dataframe.loc[row_name]
```


Exercise 3

- Read the `Orange_1.tsv`
- Print the height column
- Print the data for the tree at age 2

```
In [ ]: tree_growth = pd.read_table('../downloads/Orange_1.tsv', index_col=0)
        tree_growth
```

```
In [ ]: tree_growth = pd.read_table('../downloads/Orange_1.tsv', index_col=0)
        tree_growth
```

```
In [ ]: tree_growth.height
```

```
In [ ]: tree_growth.loc[2]
```

Many trees!

Orange.tsv

Tree	age	circumference
1	118	30
1	484	58
1	664	87
1	1004	115
...		
2	118	33
2	484	69
...		

```
In [ ]: tree_growth = pd.read_table('../downloads/Orange.tsv', index_col=0)
        tree_growth
```

In []: `tree_growth.index`

In []: `tree_growth.columns`

In []:

```
tree_growth.age
```

In []: tree_growth.age

In []: tree_growth.age.values

In []: `tree_growth.age.unique()`

In []: `tree_growth.age.unique()`

Works like a normal list:

```
In [ ]: tree_growth.age.unique()
```

Works like a normal list:

```
In [ ]: tree_growth.age.unique()[0]
```

```
In [ ]: tree_growth.age.unique()
```

Works like a normal list:

```
In [ ]: tree_growth.age.unique()[0]
```

```
In [ ]: len(tree_growth.age.unique())
```

Columns

`dataframe.columnname`

Methods:

`.max(), .min(), unique(), .values, .mean(), .sum()...`

Selecting parts of the table

Selecting parts of the table

```
In [ ]: tree_growth.circumference  # selecting a column
```

Selecting parts of the table

```
In [ ]: tree_growth.circumference  # selecting a column
```

```
In [ ]: tree_growth.loc[2]  # selecting rows with index 2
```

Selecting parts of the table

```
In [ ]: tree_growth.circumference  # selecting a column
```

```
In [ ]: tree_growth.loc[2]  # selecting rows with index 2
```

```
tree_growth.loc[ criteria ]  # select all rows that fullfills this criteria
```


Selecting parts of the table

Find the data points where the tree is younger than 200 years!

- Find *rows* => use `tree_growth.loc[]`
- Select these based on the value of column *age* => `tree_growth.age`

In []: `# The answer...`

Selecting parts of the table

Find the data points where the tree is younger than 200 years!

- Find *rows* => use `tree_growth.loc[]`
- Select these based on the value of column *age* => `tree_growth.age`

```
In [ ]: # The answer...
```

```
In [ ]: young = tree_growth.loc[tree_growth.age < 200]
        young
```

Exercise 4

- Read the data `Orange.tsv`
- Find the maximum circumference
- What tree reached that circumference, and how old was it at that time?

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

```
In [ ]: import pandas as pd  
        tree_growth = pd.read_table('../downloads/Orange.tsv', index_col=0)
```

```
In [ ]: import pandas as pd  
        tree_growth = pd.read_table('../downloads/Orange.tsv', index_col=0)
```

```
In [ ]: max_c = tree_growth.circumference.max()  
        print(max_c)
```

```
In [ ]: import pandas as pd  
        tree_growth = pd.read_table('../downloads/Orange.tsv', index_col=0)
```

```
In [ ]: max_c = tree_growth.circumference.max()  
        print(max_c)
```

```
In [ ]: tree_growth.loc[tree_growth.circumference == max_c]
```

Plotting

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

Plotting

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

```
In [ ]: orange_1 = pd.read_table('../downloads/Orange_1.tsv')  
orange_1.circumference.plot()
```


Plotting

What if no plot shows up?

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

or

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

Plotting - many trees

Plotting - many trees

- Plot a bar chart

Plotting - many trees

- Plot a bar chart

In []: `tree_growth.plot(kind='bar')`

In []: `tree_growth.plot(kind='bar', figsize=(12, 12), fontsize=12)`

Plotting

- Plot a line graph

Plotting

- Plot a line graph

```
In [ ]: # Starting with tree number 1  
tree1 = tree_growth.loc[1]
```

```
In [ ]: tree1
```

Plotting

- Plot a graph: `dataframe.plot(kind="line", x=..., y=...)`

In []: `tree1.plot(x='age', y='circumference', fontsize=14, figsize=(12,10))`

Plotting

- Plot a graph: `dataframe.plot(kind="line", x="..", y="...")`

Plotting

- Plot a graph: `dataframe.plot(kind="line", x="..", y="...")`

Let's plot all the trees!

```
In [ ]: tree_growth.plot(kind='line', x='age', y='circumference', figsize=(12, 10), fontsize=14)
```

```
In [ ]: tree_growth.plot(kind='line', x='age', y='circumference', figsize=(12, 10), fontsize=14)
```

```
:(
```

Plotting

- Plot a graph: `dataframe.plot(kind="scatter", x="..", y="...")`

In []: `tree_growth.plot(kind='scatter', x='age', y='circumference', figsize=(12, 10), fontsize=14)`

Plotting

What about the lines?

Plotting

What about the lines?

- Group the table by the index (make subtrees)
- Get one board to plot all the lines
- Draw them one by one

Plotting

What about the lines?

- Group the table by the index (make subtrees)
- Get one board to plot all the lines
- Draw them one by one

```
dataframe.groupby([what])
```

Plotting

What about the lines?

- Group the table by the index (make subtrees)
- Get one board to plot all the lines
- Draw them one by one

```
dataframe.groupby([what])
```

```
import matplotlib.pyplot as plt
```

```
fig, ax = plt.subplots()
```

Plotting

Plotting

```
In [ ]: import matplotlib.pyplot as plt

fig, ax = plt.subplots()

for index, subtree in tree_growth.groupby(['Tree']):
    subtree.plot(x='age', y='circumference', ax=ax, kind='line', fontsize=14, figsize=(12,10))
```

Exercise 5

- Use Pandas to read IMDB
- Explore it by making graphs