WORKING WITH THE COBRAPY PACKAGE (AND PANDAS)

General

`import cobra`	To use the COBRApy package, you need to import it at the top of your script.

`from cobra.io import read_sbml_model`
`model = read_sbml_model(“file.xml”)`
or
`model = cobra.io.read_sbml_model(“file.xml”)`
	Some functions come from a sub-module of the COBRApy package. You can either import these functions at the top of your script or refer to the sub-module when using the function.

Terminology	
object	Python is an object-oriented programming language and most things in Python are objects: strings, integers, floats, lists, dictionaries, etc. In COBRApy, you will encounter the objects model, reaction, metabolite, and others. Objects (can) have object-specific attributes (properties) and object-specific functions (methods).
property	An object can have different properties. In COBRApy, for example, `model` objects have (among others) the properties `model.reactions`, `model.metabolites`, and `model.genes`.
method	An object can have different methods (behaviours). You may already know the list method `list_name.append()` or the string method `string_name.strip()`. (Note that methods have brackets, properties do not.)
function	A function describes a behaviour that is not tied to an object, although it can require specific objects as arguments. An example of a function is `len()`, which can take many different classes of objects as an argument. Functions return some value.

COBRApy

Creating Metabolic Models

`from cobra import Model, Reaction, Metabolite`	To create a model from scratch, import the classes `Model`, `Reaction` and `Metabolite` at the top of your script.
`model = Model(“name_of_your_model”)`	`Model` creates a metabolic model object with the variable name `model`. `name_of_your_model` is the name of the file in which your model will be stored.
`reaction1 = Reaction(“R1”)`	`Reaction` creates a reaction object with the variable name `reaction1`. `R1` is the reaction’s `id`, a short name used to refer to the reaction, e.g., after the model is loaded into a new script, using `model.reactions.get_by_id(“R1”).`
`metabolite1 = Metabolite(“M1”)`	`Metabolite` creates a reaction object with the variable name `metabolite1`. `M1` is the metabolite’s `id`, a short name used to refer to the metabolite, e.g., after the model is loaded into a new script, using `model.metabolites.get_by_id(“M1”).`

Using Metabolic Models

`model = cobra.io.read_sbml_model(“model_file.xml”)`	This command loads a model into a new script. Any changes you make to the model within the script will not be saved to the file (although they can be, should you choose to do so).

Model Properties

`model.reactions`
`model.reactions.get_by_id(“R1”)`
	This is a DictList of all the reactions (i.e., reaction objects) in the model. In many ways, you can use it like a list, iterate over it, determine its length etc.
	Using the `get_by_id` method, a specific reaction object can be accessed using its `id`. It can then be treated like any other reaction object, e.g., its lower bound can be accessed using `model.reactions.get_by_id(“R1”).lower_bound`.

`model.exchanges`	A model has a property `exchanges`, a DictList of the model’s exchange/external reactions. COBRApy uses reaction annotations and heuristic methods to find the correct reactions. This can be imperfect, but typically works well.

`model.medium`	This is a dictionary of all exchange reactions whose bounds allow the organism to consume the respective metabolite.

`model.metabolites`
`model.metabolites.get_by_id("nadh")`
`model.metabolites.nadh.summary()`
	This is a DictList of all the metabolites (i.e., metabolite objects) in the model. In many ways, you can use it like a list, iterate over it, determine its length etc.
	You can access a specific metabolite either using the `get_by_id` method or just using its `id` as a property (see left). It can then be treated like any other metabolite object, e.g., to look at its `summary()`.

`model.genes`	This is a DictList of all the genes (i.e., gene objects) in the model.
`model.objective = "REACTION ID"`	You can look at the model's objective reaction – the reaction whose flux FBA will optimise – or set a new objective using the respective reaction's ID.
Model Methods	

`solution = model.optimize()`
`solution.objective_value`
`solution.fluxes`
	The `optimize()` method performs FBA on the respective model. The `solution` object contains useful information about the FBA result, such as the flux through the objective reaction (`solution.objective_value`) and a pandas Series containing the fluxes through all of the reactions in the model (`solution.fluxes`).

`summary = model.summary()`	The model's `summary()` method gives you information about the outcome of the most recent FBA. It is mostly useful to print as output; accessing information within it is difficult.
Functions	

`from cobra.flux_analysis import single_reaction_deletion`
`srd_solution = single_reaction_deletion(model,`
`reaction_list=["R4", "R7", "R19"])`
	The `single_reaction_deletion` function deletes each reaction individually from the full model and finds the optimal growth under that condition. The `srd_solution` is a pandas DataFrame listing the ids of the deleted reactions, the corresponding optimal growth and the solver's status.
	The function accepts an argument `reaction_list` with reaction IDs. If passed, the single reaction deletion is only performed for those reactions.
	Note: The column "ids" contains the reaction IDs as the datatype set rather than as strings, but you can easily convert a set into a string using `str(set_name)`.

`from cobra.flux_analysis import flux_variability_analysis`
`fva_solution = flux_variability_analysis(model,`
`reaction_list= , fraction_of_optimum= )`
	The `flux_variability_analysis` function finds the minimal and the maximal flux through each reaction in any FBA solution without changing the flux through the objective reaction. The `fva_solution` is a pandas DataFrame with an index of the reaction IDs and two columns “minimum” and “maximum” listing the respective flux values.
	The function accepts an argument `reaction_list` with reaction IDs. If passed, flux variability analysis is only performed for those reactions. It also accepts an argument `fraction_of_optimum`, which allows you to perform FVA under the condition that the flux through the objective reaction is at least a given fraction of its optimum. The default is 1.

Reactions

Reaction Properties

`reaction.id = “ALKP”`	A reaction’s ID is a short name which can be used to quickly find information about the respective reaction in various contexts.
`reaction.name = “Alkaline Phosphatase”`	The name of the reaction is typically longer and can be quite descriptive. It is often the name of the enzyme which catalyses the reaction.

`print(reaction.lower_bound)`
`reaction.upper_bound = 20`
	Reactions have a lower bound and upper bound, which define the range of possible flux values the reaction can have in the FBA solution.

`reaction.metabolites`	The `metabolites` property is a dictionary with all the metabolite objects involved in the reaction as keys and their stoichiometric coefficients as values.
`reaction.reactants` `reaction.products`	The `reactants` and `products` properties are lists containing all metabolite objects with negative or positive stoichiometric coefficients, respectively.
`reaction.notes`	Some reactions will have a `.notes` property, a dictionary containing additional information. For example, you may be able to find information about the subsystem a reaction is part of using `reaction.notes[“SUBSYSTEM”]`.

Metabolites

Metabolite Properties

`metabolite.id` `metabolite.name`	Like reactions, each metabolite has a (short) `id` and can have a (longer) `name` property. When you `print()` the metabolite object itself, it will automatically return `metabolite.id`.
`metabolite.compartment`	A metabolite can be assigned to a given compartment. Typical compartments are within the cell (`“c”`) and outside the cell (`“e”`).
`metabolite.formula`	The `formula` property is the metabolite’s chemical formula as a string. For ethanol, it would be `“C2H6O1”`.
`metabolite.elements`	The `elements` property is a dictionary with the elements as keys and the number of atoms as values. For ethanol, it would be `{“C”: 2, “H”: 6, “O”: 1}`.

Using pandas

The output of certain COBRApy functions is a pandas Series or a pandas DataFrame. You can obviously create these two data structures yourself as well, but in the context of COBRApy, you mostly need to be able to access the information stored in them. Here, you will find a couple of useful ways to do that.

`import pandas as pd`	By convention, the pandas package is usually imported as pd.

Using pandas Series

A pandas Series is a data structure which is technically one-dimensional (one column), but each entry (row) also has a label, which can be used to refer to the respective entry. The “list of labels” of all entries is called the index.

`s = pd.Series([1, 3, 5, 7, 6, 8])` `s = pd.Series([1, 3, “apples”, 7, 6.4, 8])` `s = pd.Series({“apple”: 2.4, “pear”: 3.1, “orange”: 5.7, “durian”: -100000., “lychee”: 9.9})`	A simple Series can be created from a list. The list can contain entries of multiple data types (integers, strings, floats, …). The default index is a range numbering the entries starting from 0. A Series can also be created from a dictionary, in which case the keys become the labels/the index and the values become the entries.
`s.index`	You can access the Series’ index (to print it, for example).

```
s[3:5]
```

You can access part of the Series using slicing. This example will return entries 3 and 4, as the end of the slice is non-inclusive. You can also use slicing if your index is not a range, in which case the integers simply refer to the entries' positions.

```
s.loc[“pear”] # (equivalent to s[“pear”])
s.loc[[“apple”, “durian”]]
s.loc[“apple”:“orange”]
```

`.loc` is primarily label-based. You can access an individual entry by passing its label, which will return only the entry (here: 3.1). You can also access a selection of entries by passing a list of labels or using “label slicing” (Label slicing includes the end point!). Note that the latter two will return the entries WITH their labels, i.e., a (shorter) pandas Series.

```
s.iloc[1]
s.iloc[[0, 3]]
s.iloc[:3]
```

`.iloc` is primarily position-based. You can access an individual entry by passing its position in the Series, which will return only the entry (here: 3.1). You can also access a selection of entries by passing a list of positions or using slicing. Note that the latter two will return the entries WITH their labels, i.e., a smaller pandas Series.

```
s[s > 5]
```

Selecting by condition (“Boolean indexing”) enables us to find entries that meet a given condition and returns a (shorter) pandas Series. In this case, we are selecting all entries larger than 5.

```
for label, entry in s.iteritems():
      print(label, entry)
```

Finding rows where the *label* meets a given condition can unfortunately not be done easily with Boolean indexing. For those cases, you may find it easier to iterate over the Series using `.iteritems()`. The approach on the left allows you to access each row's label and entry individually.

Using the pandas DataFrame

A pandas DataFrame is a two-dimensional data structure with multiple columns. A row can be referred to using its row label, an entry can be referred to with a row label and a column label. The “list of labels” of all entries is called the index.

```
data = {“species”: [“dog”, “fish”, “phoenix”, “dragon”],
        “real”: [True, True, False, False],
        “interesting”: [“quite”, “not”, “very”, “too”],
        “cost”: [400, -20, 9000, -1000000]}
df = pd.DataFrame(data, index=[“Odie”, “Pudge”, “Fawkes”,
“Bob”])
```

A pandas DataFrame can be created in various ways; one is shown on the left as an example. The dictionary `data` defines the DataFrame's columns – the keys are the column labels, the values (lists) are the column content – and the `index` argument in the `pd.DataFrame()` function defines the row labels. If the `index` argument is not used, the row labels are set to a range starting at 0.

```
df.index
```

You can access the DataFrame's index (to print it, for example).

`df[1:3]`	You can access part of the DataFrame using slicing. This example will return entries 1 and 2, as the end of the slice is non-inclusive. You can also use slicing if your index is not a range, in which case the integers simply refer to the rows' positions.

`df[“species”]`
`df[[“species”, “interesting”]]`
	You can access a selection of columns by indexing, using either a column label or a list of column labels.

`df.loc[row_labels]`
`df.loc[row_labels, column_labels]`
 	
`df.loc[[“Odie”, “Bob”]]`
`df.loc[:, “cost”]`
`df.loc[“Pudge”, “real”:“interesting”]`
	`.loc` is primarily label-based. You can see the general two options in the first two lines. The labels of choice for both the rows and the columns can be specified with a single colon (`:`) if you want to choose all, slicing (Label slicing is inclusive!), an individual label or a list of labels. Depending on your choices, the result is a (shorter) DataFrame, a Series, or an individual entry.

`df.iloc[row_ positions]`
`df.iloc[row_ positions, column_positions]`
 	
`df.iloc[[0, 3]]`
`df.iloc[:, 1]`
`df.iloc[1, 1:3]`
	`.iloc` is primarily position-based. You can see the general two options in the first two lines. The positions of choice for both rows and columns can be specified with a single colon (`:`) if you want to choose all, slicing (Regular slicing is non-inclusive!), an individual position or a list of positions. Depending on your choices, the result is a (shorter) DataFrame, a Series, or an individual entry.

`df[df[“real”] == False]`
`df[abs(df[“cost”]) < 500]`
`df[(df[“real”] == False) & (abs(df[“cost”]) < 500)]`
	Selecting by condition (“Boolean indexing”) allows us to find the entries that meet a given condition. In the first example on the left, we can find all animals which are not real (where the entry in the `“real”` column is `False`). In the second example, we can find all animals that are moderately costly or lucrative (where the absolute value in the `“cost”` column is less than 500 – and before you start arguing that you don't have to pay people to take a fish off your hands: Yes, you do. Fish are boring.). Useful operators for Boolean indexing are:
	`< / >` smaller than / larger than
	`<= / >=` smaller than or equal to / larger than or equal to
	`== / !=` equal to / not equal to
	You can also combine multiple conditions in the form `df[(condition1) & (condition2)]`. You can see an example on the left. Useful operators are:
	`&` if both conditions must be met (“and”)
	`|` if at least one condition must be met (“or”)

`df[“species”].tolist()`	A DataFrame column can be converted to a list using the `.tolist()` method. The entries of the list may not be of the type you expect (string, float, …), so you may have to convert the entries before working with them.

`for index, series in df.iterrows():`
`    print(index, series[“cost”])`
	You can iterate over a DataFrame using `.iterrows()`.The approach on the left allows you to access each row’s label and the column entries as a pandas Series individually. You can access the value in each column using its label.