Lecture 1 Code Workshop

5 February 2024

We first begin by importing necessary packages. These will allow us to access and view our data.

```
In [2]: import pandas as pd
        import urllib
        import numpy as np
```

We will ask our computer to run code in these packages by "calling" them (typing out their name). For convenience, package names can be abbreviated. Typically, numpy is called as "np" and pandas as "pd", but these abbreviations are entirely up to you. The package urllib is used to access the internet and retrieve the spreadsheet (dataframe) that we will be looking at from Abby Doyle's Github repository: 'https://raw.githubusercontent.com/doylelab/rxnpredict/master/data_table.csv'.

Sometimes we don't want to import the entire package with all of its code. We just want a little bit of it. To do that, we specify:

```
In [3]: from rdkit import Chem
```

Here, we are only importing code and functions that's associated with Chem.

It's not important to know every package in python, and there are too many to really keep track of. Below is a helpful table.

Package Name	***Typical Abbreviation***	***Usage***
numpy	np	Mathematical operations (including things like logs and square roots) and the creation of vectors.
pandas	pd	Importing and saving spreadsheet data. Looking up values in spreadsheets quickly.
sklearn	N/A	Basic machine learning functions. Data standardization and preprocessing. Common example datasets.
rdkit	N/A	Working with molecules - drawing them, turning them into fingerprints etc.

torch	N/A	Deep learning functions. Attempts to keep notation consistent with numpy notation.

We will now go and get our dataset from Abby Doyle's Github repository and saving it as "doyle.csv".

```
In [5]: github_repo = 'https://raw.githubusercontent.com/doylelab/rxnpredict/master/data_table.csv'

        urllib.request.urlretrieve(github_repo, 'doyle.csv')

Out[5]: ('doyle.csv', <http.client.HTTPMessage at 0x7f8640487c90>)
```

We will then load this comma separated values (CSV) file with pandas by asking the pandas package to run a piece of its code called "read_csv()". We specify this with the "." after the package name.

```
In [6]: df = pd.read_csv('doyle.csv')
```

For questions about what to put into the brackets of functions, please refer to the official documentation. Other than rdkit, the standard packages are often well documented (rdkit's documentation can be hit or miss). If rdkit's documentation is lacking, I would recommend asking Claude (https://claude.ai).

We can see the spreadsheet, which we often refer to as a "dataframe" in programming by simply typing the name of our dataframe, "df", which we named above.

We can see the first few rows.

```
In [8]: df.head()
```

Out[8]:

	plate	row	col	base	base_cas_number	base_smiles	ligand	ligand_cas_number	ligand_smiles	aryl_halide_nu
0	1	1	1	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	
1	1	1	2	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	
2	1	1	3	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	
3	1	1	4	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	
4	1	1	5	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	

click to scroll output; double click to hide

Or the first 3 rows.

```
In [9]: df.head(3)
```

Out[9]:

	plate	row	col	base	base_cas_number	base_smiles	ligand	ligand_cas_number	ligand_smiles	aryl_halide_nu
0	1	1	1	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	
1	1	1	2	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	
2	1	1	3	P2Et	165535-45-5	CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC	XPhos	564483-18-7	CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...	

Or the last few rows.

```
In [10]: df.tail()
```

4594	3	32	44	MTBD	84030-20-6	CN1CCCN2CCCN=C12	AdBrettPhos	1160861-59-5	CC(C1=C(C2=C(OC)C=CC(OC)=C2P(C34CC5CC(C4)CC(C5.
4595	3	32	45	MTBD	84030-20-6	CN1CCCN2CCCN=C12	AdBrettPhos	1160861-59-5	CC(C1=C(C2=C(OC)C=CC(OC)=C2P(C34CC5CC(C4)CC(C5.
4596	3	32	46	MTBD	84030-20-6	CN1CCCN2CCCN=C12	AdBrettPhos	1160861-59-5	CC(C1=C(C2=C(OC)C=CC(OC)=C2P(C34CC5CC(C4)CC(C5.
4597	3	32	47	MTBD	84030-20-6	CN1CCCN2CCCN=C12	AdBrettPhos	1160861-59-5	CC(C1=C(C2=C(OC)C=CC(OC)=C2P(C34CC5CC(C4)CC(C5.
4598	3	32	48	MTBD	84030-20-6	CN1CCCN2CCCN=C12	AdBrettPhos	1160861-59-5	CC(C1=C(C2=C(OC)C=CC(OC)=C2P(C34CC5CC(C4)CC(C5.

Or the last 4 rows.

```
In [11]: df.tail(4)
```

Out[11]:

	plate	row	col	base	base_cas_number	base_smiles	ligand	ligand_cas_number	ligand_smile
4595	3	32	45	MTBD	84030-20-6	CN1CCCN2CCCN=C12	AdBrettPhos	1160861-59-5	CC(C1=C(C2=C(OC)C=CC(OC)=C2P(C34CC5CC(C4)CC(C5.
4596	3	32	46	MTBD	84030-20-6	CN1CCCN2CCCN=C12	AdBrettPhos	1160861-59-5	CC(C1=C(C2=C(OC)C=CC(OC)=C2P(C34CC5CC(C4)CC(C5.
4597	3	32	47	MTBD	84030-20-6	CN1CCCN2CCCN=C12	AdBrettPhos	1160861-59-5	CC(C1=C(C2=C(OC)C=CC(OC)=C2P(C34CC5CC(C4)CC(C5.
4598	3	32	48	MTBD	84030-20-6	CN1CCCN2CCCN=C12	AdBrettPhos	1160861-59-5	CC(C1=C(C2=C(OC)C=CC(OC)=C2P(C34CC5CC(C4)CC(C5.

We can see the column names.

```
In [19]: df.keys()
Out[19]: Index(['plate', 'row', 'col', 'base', 'base_cas_number', 'base_smiles',
                'ligand', 'ligand_cas_number', 'ligand_smiles', 'aryl_halide_number',
                'aryl_halide', 'aryl_halide_smiles', 'additive_number', 'additive',
                'additive_smiles', 'product_smiles', 'yield'],
               dtype='object')
```

And search the dataframe very quickly. There are two main ways to search pandas dataframes. The first is using "iloc" and the second is using "loc". Although similar in name and function, "iloc" pulls out specific numbered rows and "loc" allows you to search with conditions. For the most part, "loc" is the more versatile command and the one you will most like use more often.

To pull out the 12th row, we can do:

```
In [12]: df.iloc[12]
```

```
Out[12]: plate                                                          1
         row                                                            1
         col                                                           13
         base                                                        P2Et
         base_cas_number                                      165535-45-5
         base_smiles                  CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC
         ligand                                                     XPhos
         ligand_cas_number                                    564483-18-7
         ligand_smiles        CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...
         aryl_halide_number                                          13.0
         aryl_halide                                      3-chloropyridine
         aryl_halide_smiles                                     Clc1cccnc1
         additive_number                                              NaN
         additive                                                     NaN
         additive_smiles                                              NaN
         product_smiles                                Cc1ccc(Nc2cccnc2)cc1
         yield                                                    49.059786
         Name: 12, dtype: object
```

Notice that the same command with "loc" also works.

```
In [13]: df.loc[12]
```

```
Out[13]: plate                                                          1
         row                                                            1
         col                                                           13
         base                                                        P2Et
         base_cas_number                                      165535-45-5
         base_smiles                  CN(C)P(N(C)C)(N(C)C)=NP(N(C)C)(N(C)C)=NCC
         ligand                                                     XPhos
         ligand_cas_number                                    564483-18-7
         ligand_smiles        CC(C)C1=CC(C(C)C)=CC(C(C)C)=C1C2=C(P(C3CCCCC3)...
         aryl_halide_number                                          13.0
         aryl_halide                                      3-chloropyridine
         aryl_halide_smiles                                     Clc1cccnc1
         additive_number                                              NaN
         additive                                                     NaN
         additive_smiles                                              NaN
         product_smiles                                Cc1ccc(Nc2cccnc2)cc1
         yield                                                    49.059786
         Name: 12, dtype: object
```

If we want to find all the rows that had yields over 80%:

```
In [15]: df.loc[df['yield'] > 80]
```

mber	aryl_halide	aryl_halide_smiles	additive_number	additive	additive_smiles	product_smiles	yield
11.0	2-bromopyridine	Brc1ccccn1	NaN	NaN	NaN	Cc1ccc(Nc2ccccn2)cc1	82.686578
12.0	2-iodopyridine	Ic1ccccn1	NaN	NaN	NaN	Cc1ccc(Nc2ccccn2)cc1	81.209845
14.0	3-bromopyridine	Brc1cccnc1	NaN	NaN	NaN	Cc1ccc(Nc2cccnc2)cc1	80.415221
15.0	3-iodopyridine	Ic1cccnc1	NaN	NaN	NaN	Cc1ccc(Nc2cccnc2)cc1	87.277323
8.0	1-bromo-4-ethylbenzene	CCc1ccc(Br)cc1	NaN	NaN	NaN	CCc1ccc(Nc2ccc(C)cc2)cc1	83.761103
...	...	...	...	...	...	...	...
12.0	2-iodopyridine	Ic1ccccn1	18.0	N,N-dibenzylisoxazol-5-amine	C(N(Cc1ccccc1)c2oncc2)c3ccccc3	Cc1ccc(Nc2ccccn2)cc1	83.496024
12.0	2-iodopyridine	Ic1ccccn1	20.0	5-methyl-3-(1H-pyrrol-1-yl)isoxazole	Cc1onc(c1)n2cccc2	Cc1ccc(Nc2ccccn2)cc1	83.442335

Notice that "iloc" yields an error - it is not the right command for this style of searching.

```
In [16]: df.iloc[df['yield'] > 80]
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
/var/folders/07/jy3d68h93vx9mzcrm251lhy00000gn/T/ipykernel_4777/1366834652.py in <module>
----> 1 df.iloc[df['yield'] > 80]

~/miniconda3/envs/ccdc3.7/lib/python3.7/site-packages/pandas/core/indexing.py in __getitem__(self, key)
    929
    930             maybe_callable = com.apply_if_callable(key, self.obj)
--> 931             return self._getitem_axis(maybe_callable, axis=axis)
    932
    933     def _is_scalar_access(self, key: tuple):

~/miniconda3/envs/ccdc3.7/lib/python3.7/site-packages/pandas/core/indexing.py in _getitem_axis(self, key, axis)
   1550
   1551         if com.is_bool_indexer(key):
-> 1552             self._validate_key(key, axis)
   1553             return self._getbool_axis(key, axis=axis)
   1554

~/miniconda3/envs/ccdc3.7/lib/python3.7/site-packages/pandas/core/indexing.py in _validate_key(self, key, axis)
   1394                 if key.index.inferred_type == "integer":
   1395                     raise NotImplementedError(
-> 1396                         "iLocation based boolean "
   1397                         "indexing on an integer type "
   1398                         "is not available"

NotImplementedError: iLocation based boolean indexing on an integer type is not available
```

We can ask to see all of the products that had yields over 80%:

```
In [31]: df.loc[df['yield'] > 80, ['product_smiles']]
```

Out[31]:

	product_smiles
218	Cc1ccc(Nc2ccccn2)cc1
219	Cc1ccc(Nc2ccccn2)cc1
221	Cc1ccc(Nc2cccnc2)cc1
222	Cc1ccc(Nc2cccnc2)cc1
231	CCc1ccc(Nc2ccc(C)cc2)cc1
...	...
4498	Cc1ccc(Nc2ccccn2)cc1
4514	Cc1ccc(Nc2ccccn2)cc1
4545	Cc1ccc(Nc2ccccn2)cc1
4546	Cc1ccc(Nc2ccccn2)cc1
4594	Cc1ccc(Nc2ccccn2)cc1

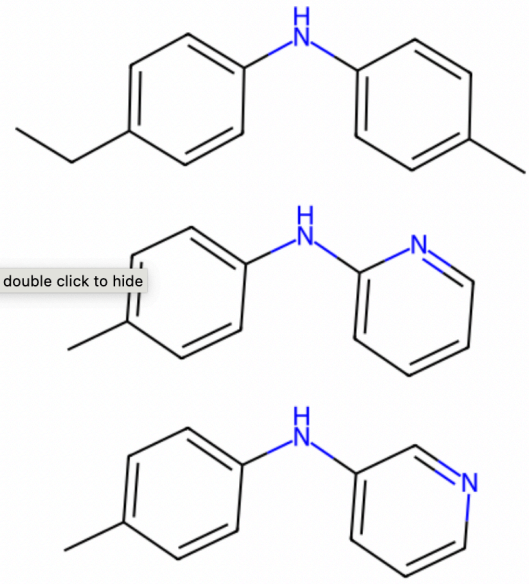
312 rows × 1 columns

That isn't so useful. We don't want duplicates of names, just one of each. We can do that with:

```
In [29]: np.unique(df.loc[df['yield'] > 80, ['product_smiles']])
Out[29]: array(['CCc1ccc(Nc2ccc(C)cc2)cc1', 'Cc1ccc(Nc2ccccn2)cc1',
                'Cc1ccc(Nc2cccnc2)cc1'], dtype=object)
```

Okay... How do we visualize these compounds? That can be done like so with rdkit.

```
In [32]: for smiles in np.unique(df.loc[df['yield'] > 80, ['product_smiles']]):
             mol = Chem.MolFromSmiles(smiles)
             display(mol)
```



In this section of code, first, we take each one of our product SMILES strings and turn it into an rdkit molecule. Then, we display it.

Alright, what if we want to find the yields for reactions that used AdBrettPhos as the catalyst but NOT P2Et as the base? We specify each condition like so: "(condition 1) & (condition 2)". The does not equal sign is "!=".

```
In [34]: df.loc[(df['ligand'] == 'AdBrettPhos') & (df['base'] != "P2Et"), ['yield']]
```

Out[34]:

	yield
592	25.142084
593	40.991093
594	45.960477
595	1.429259
596	39.815220
...	...
4594	86.233157
4595	1.440081
4596	43.538365
4597	69.795902
4598	0.000000

767 rows × 1 columns

Notice that if we do not have "==" but only "=", we get an error again. This is because we want to ask if the condition has been met (the ligand's name is AdBrettPhos), not setting a variable to be that condition.

```
In [35]: df.loc[(df['ligand'] = 'AdBrettPhos') & (df['base'] != "P2Et"), ['yield']]
  File "/var/folders/07/jy3d68h93vx9mzcrm251lhy00000gn/T/ipykernel_4777/3435545176.py", line 1
    df.loc[(df['ligand'] = 'AdBrettPhos') & (df['base'] != "P2Et"), ['yield']]
                         ^
SyntaxError: invalid syntax
```

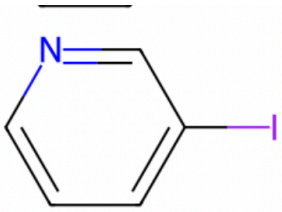
What is the average yield of these reactions?

```
In [37]: np.mean(df.loc[(df['ligand'] == 'AdBrettPhos') & (df['base'] != "P2Et"), ['yield']])
Out[37]: yield    34.339779
         dtype: float64
```

Let's take a look at our starting materials now. I've defined a function called see_molecules(). Python programming is somewhat out of the scope of this course, but just know that this will allow me to visualize the molecules like we did above. I've now just made it into a function so that I don't have to write it out so many times.

```
In [40]: def see_molecules(smiles):
             mol = Chem.MolFromSmiles(smiles)
             return display(mol)

         for smiles in df['aryl_halide_smiles'].unique():
             see_molecules(smiles)
```




```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/07/jy3d68h93vx9mzcrm251lhy00000gn/T/ipykernel_4777/3561165102.py in <module>
      4
      5 for smiles in df['aryl_halide_smiles'].unique():
----> 6     see_molecules(smiles)

/var/folders/07/jy3d68h93vx9mzcrm251lhy00000gn/T/ipykernel_4777/3561165102.py in see_molecules(smiles)
      1 def see_molecules(smiles):
----> 2     mol = Chem.MolFromSmiles(smiles)
      3     return display(mol)
      4
      5 for smiles in df['aryl_halide_smiles'].unique():

TypeError: No registered converter was able to produce a C++ rvalue of type std::__1::basic_string<wchar_t, std::__
1::char_traits<wchar_t>, std::__1::allocator<wchar_t> > from this Python object of type float
```

Oh no! What's happened?! Well, rdkit does NOT like NAN entries and this error is almost always caused by trying to convert an empty cell in a spreadsheet to a molecular image. Not to fear, we can fix this pretty simply:

In [41]:
```
def see_molecules(smiles):
    if pd.isna(smiles) == False:
        mol = Chem.MolFromSmiles(smiles)
        return display(mol)
    else:
        return ''

for smiles in df['aryl_halide_smiles'].unique():
    see_molecules(smiles)
```

For fun, could you try this piece of code to visualize the bases, ligands, and additives?

– Until Next Time!