Python's `magic` functions

Disclaimer: These notes and examples are an adaptation of the references listed at the end. They are compiled to fit the scope of this specific course.

Introduction

By now, you all have experienced commands such as `%whos` or `%load`. These are two examples of `magic functions` or `magics`.

According to python.org:

> The magic function system provides a series of functions which allow you to control the behavior of python itself, plus a lot of system-type features.

`magics` are pre-defined functions that utilize command-line-like syntax and are prefaced with a `%` or `%%`.

There are two kinds of `magics`, line-oriented and cell-oriented:

- `line magics` are prefixed with the `%` character, such as `%whos` and `%load`. `line magics` are written in just one line. For example, this will time the given statement:

  > ```
  > %timeit sum(range(10000))
  > ```
  >
  > as you can see both the magic function `%timeit` and the argument `sum(range(10000))` are in the same line.

- `cell magics` are prefixed with a `%%`. They are typically used once per cell. `cell magics` can spread over multiple lines. Here is an example

  > ```
  > %%timeit
  >
  > sum_1 = 0
  > for counter in range(20000):
  >     sum_1 += counter
  > print(sum_1)
  > ```
  >
  > will time the execution of the given lines following `%%timeit`.

Note: Argument(s) are passed to `magics` without parentheses, quotes, or commas.

Hint 1: Run `%magic` in a JupyterLab cell to get a full list of all available magic functions along with their documentation. The output will have a heading like:

```
%magic

IPython's 'magic' functions
===========================

The magic function system provides a series of functions which allow you to
control the behavior of IPython itself, plus a lot of system-type
features. There are two kinds of magics, line-oriented and cell-oriented.
```

The rest of this handout will walk you through the most commonly used `magics` and examples. For complete syntax and parameters of these functions, please consult https://ipython.readthedocs.io/en/stable/interactive/magics.html

`%quickref` to the rescue

Besides `%magic`, one of the most common magic functions is `%quickref`. This will bring information about JupyterLab and its magic functions. The heading of the output will look like this:

```
%quickref

IPython -- An enhanced Interactive Python - Quick Reference Card
================================================================

obj?, obj??      : Get help, or more help for object (also works as
                   ?obj, ??obj).
?foo.*abc*       : List names in 'foo' containing 'abc' in them.
%magic           : Information about IPython's 'magic' % functions.
```

This quick reference card displays the most widely used commands and available magic functions. As an example, using `?` or `??` before or after an object will provide help or more help with that object. Try the following:

```
str_1 = 'sample text'
?str_1  # as an alternative use ??str_1 or str_1? or str_1??
```

Commonly used `line magics`

These are the commonly used `line magics` along with several practical examples:

`%magic` & `%lsmagic`

`%magic` provides a complete guide on the current magics. This is your best reference on `magics`.

Output of `%magic` can be overwhelmingly long and elaborate. `%lsmagic` provides just a list of `magics` grouped into `line` and `cells` as

```
%lsmagic

▼ root:
  ▶ line:
  ▶ cell:
```

Opening each of these two subcategories will provide all the commands available to you. Browse the list and if you want to know more about a specific magic function, call the function followed by a question mark, like this `%dhist?`. You will be provided with handy documentation

detailed to that function.

```
%dhist?
```

```
Docstring:
Print your history of visited directories.

%dhist       -> print full history
%dhist n     -> print last n entries only
%dhist n1 n2 -> print entries between n1 and n2 (n2 not included)

This history is automatically maintained by the %cd command, and
always available as the global list variable _dh. You can use %cd -<n>
to go to directory number <n>.
```

`%who`, `%whos`, & `who_ls` for listing variables

- `%who` lists all the variables with the minimum information
- `%whos` is like `%who`, but it provides additional information about each variable.
- `%who_ls` provides a sorted list of all the variables.

```
import random
import math

a, b, c, x, y, z = 'banana', 3, 1.5, 'peach', 10, True

print('output of %who is')
%who

print('\noutput of %whos is:')
%whos

print('\noutput of %who_ls is:')
%who_ls
```

yields

```
output of %who is:
a   b    c    math     random  x    y    z
```

```
output of %whos is:
Variable   Type      Data/Info
-----------------------------
a          str       banana
b          int       3
c          float     1.5
math       module    <module 'math' (built-in)>
random     module    <module 'random' from 'C:<...>aconda3\\lib\\random.py'>
x          str       peach
y          int       10
z          bool      True

output of %who_ls is:
['a', 'b', 'c', 'math', 'random', 'x', 'y', 'z']
```

Note: Providing the data type after the command will report only the variables of that type. If you provide more than one data type, separate them with just one white space. Example:

```
import random
import math

a, b, c, x, y, z = 'banana', 3, 1.5, 'peach', 10, True

print('numerical (int and float) variables are:')
%whos int float

print('\nimported modules are:')
%who module

print('\nstring variables are:')
%who_ls str
```

yields

```
numerical (int and float) variables are:
Variable   Type      Data/Info
-----------------------------
b          int       3
c          float     1.5
y          int       10
```

```
imported modules are:
math   random

string variables are:
['a', 'x']
```

Practical note: I have found `%whos` a lot more useful and practical than `%who` and `%who_ls`; mainly because of the tabular organized format and detail of the provided information.

Next: `%pinfo`

Once we know a variable's name, `%pinfo object` can provide information about it. This magic function is equivalent to `object?`.

A practical obvious note: Try to get used to using `object?` notation; most of the time it is much faster than searching the internet for help! Additionally, the notation of `object?` is much more useble than `%pinfo object`.

`%precision`

Once you know what variables you want to work with or display, `%precision num_1` can set floating point precision to `num_1` digits. This is just a display option and does not change the value of the numerical variables.

```
import math

%precision 4  # 4 digits of precision
math.pi
```

yields

```
3.1416
```

Note: If no `num_1` argument is provided, defaults will be restored.

Common magics for working with directories/ history

It is not very uncommon to lose track of history of cells you have ran and location of your files and notebooks. The following are common `magics` in working with directories/ history.

- `%pwd` : standing for **`p`rint `w`orking `d`irectory** returns the current working directory path. This is particularly useful when you have multiple open notebooks in JupyterLab and want to know which folder each notebook is located. Running `%pwd` in each notebook will return that notebook's location (directory).
  - `%dhist` : Print your history of visited directories.
  - `%history` : Print input history with the most recent last. This doesn't show line numbers so it can be pasted directly into an editor. This can be useful if you want to put all of the commands executed in the recent session into one cell.

`%load`

- `%load` : Load code into the current front-end. We will use this magic function very often. Here is a few examples:

> ```
> %load 8
> ```
>
> loads snippet from cell 8
>
> ```
> %load 8-15
> ```
>
> loads snippets from cell 8 to 15

`%load_ext`, and `%unload_ext`

`%load_ext` is used for loading an extension in Jupyterlab. Similarly, `%unload_ext` is used for unloading an extension. For example, `%load_ext autoreload` will load the extension `autoreload` and `%unload_ext autoreload` will unload it.

Note: You need to install extensions before loading them!

Use `%save` for creating a `.py` file

`%save` saves a set of lines or a macro to a given filename. We will use this often. Here is an example of using this `magic` :

> ```
> %save my_code.py 1-35
> ```
>
> will save snippets in cells 1-35 in a python file with the name my_code

Use `%run` for running a `.py` and `.ipynb` files

`%run file_name` runs the python script with the extension of `.py` or a JupyterLab notebook with the extension of `.ipynb`. As an example, imagine you have `nb_1.ipynb` notebook with the following four cells

```
[ ]: list_1 = [0.1, 1, 0.1, 0.3, 0.5, 0.5, 0.8, 0.2]
     print(f'list_1 is \n{list_1}')

[ ]: def remove_element(y):
         '''filtering out values under 0.3'''
         return (y<0.3)

[ ]: from itertools import filterfalse

     filtered_list = filterfalse(remove_element, list_1)

[ ]: print(f'\nupdated list is \n{list(filtered_list)}')
```

running `%run nb_1.ipynb` in another notebook (same directory), will run all the cells in `nb_1` and print out all its outputs as in

```
[1]: %run nb_1.ipynb
     list_1 is
     [0.1, 1, 0.1, 0.3, 0.5, 0.5, 0.8, 0.2]

     updated list is
     [1, 0.3, 0.5, 0.5, 0.8]
```

Practical note: This is a handy command, especially if you want to have your code in one file and test its output/performance by running it from another file. This practice eliminates the risk of accidentally modifying your original code (in this case, code stored in `nb_1.ipynb`) while running your code.

`%time` & `%timeit`

- `%time` times execution of a Python statement or expression **once**. Here is an example

  ```
  %time sum(range(10001))
  ```

  will time adding up integer numbers from 0 to 10000. `%time` provides very basic timing functionality; use the `%timeit` magic instead. `%timeit` can do the same thing and offer more information about the run time.

  ```
  %timeit sum(range(10001))
  ```

  will time adding up integer numbers from 0 to 10000. `%timeit` runs the expression more than once and reports a more detailed execution time. `%time` and `%timeit` can be used as `cell magics`. See the next part for the details

Four `cell magics`

So far, all the discussed `magics` are `line magics`. The rest of this handout will cover the use of four cell magics `%%time`, `%%timeit`, `%%writefile`, and `%%latex`.

`%%time` & `%%timeit`

Both `%%time` and `%%timeit` could be used for the timing execution time of a cell. We will use `%%timeit` as it is more accurate.

```
%%timeit

list_1=[]
for item in range(20000):
    list_1.append(item**2)
total_1 = sum(list_1)
```

will time execution of the cell contents after `%%timeit`.

Use `%%writefile my_file_name` for writing files from a cell

We can use `%%writefile my_file_name` within a Jupyterlab cell and start writing the file contents. Here is an example:

> Run the following in one cell of a jupyterlab notebook.
>
> ```
> %%writefile test_file.py
>
> a, *b = [10, 20, 30, 40, 50, 60, 70]
> c = [item **2 for item in b if item % 3 == 0]
> for item in c:
>     print(c)
> ```
>
> creates a `test_file.py` file in the same directory as the notebook and will write the cell's contents (except `%%writefile test_file.py` line) in that file.

`%%latex` as the last and a fun cell magic

Run the following in a jupyterlab cell and enjoy the output!

```
%%latex

$$\sum_{1}^{n}{x} = \dfrac{n \cdot(n + 1)}{2}$$
```

References

This document is an adaption from the following references:

1- *Python online documentation*; available at https://docs.python.org/3/

2- *A Byte of Python*; available at https://Python.swaroopch.com/

3- *Introduction to programming in Python*; available at https://introcs.cs.princeton.edu/Python/home

4- https://ipython.readthedocs.io/en/stable/interactive/magics.html