Computational Methods in Economics (winter term 2019/20)

Lecture 1 - Introduction

In [1]:
```
# Author: Alex Schmitt (schmitt@ifo.de)

import datetime
print('Last update: ' + str(datetime.datetime.today()))
```

```
Last update: 2019-10-15 20:04:38.746563
```

In [2]:
```
import IPython.display as display
```

This Lecture

- This Course: Logistics
- What is "Computational Economics" and why do we need it?
- Python: An Overview
- Working with Jupyter Notebooks

This Course: Logistics

Who?

Alex Schmitt, Ph.D.

- Economist at the ifo Institute, Center for Energy, Climate and Resources
- PhD in Economics from Stockholm University
- Taught "Introduction to Matlab" in Stockholm, "Climate Economics" at LMU; third edition of CME in WS 2019

- Web: https://www.ifo.de/schmitt-a
- Email: schmitt@ifo.de
- Phone: 089 9224 1408
- Office Hours: On appointment!

Christina Littlejohn (in January)

- PhD Student at the ifo Institute, Center for Energy, Climate and Resources
- Web: https://www.ifo.de/littlejohn-c
- Email: littlejohn@ifo.de

- Phone: 089 9224 1433
- Office Hours: On appointment!

When and Where?

- 6 ECTS course (2h + 2h per week)
- Lecture + Tutorial: Wednesday, 14-18 c.t., A 022
- Examination: **Oral** exam, date TBA

Syllabus

W 01 (16/10): Introduction/Version Control

W 02 (23/10): Python Basics

W 03 (30/10): Python: Numpy, Pandas, OOP

W 04 (06/11): Computer Basics

W 05 (13/11): Systems of Linear Equations

W 06 (20/11): Root Finding

W 07 (27/11): Numerical Optimization I

W 08 (04/12): Numerical Optimization II

W 09 (11/12): Function Approximation I

W 10 (18/12): Function Approximation I

W 11 (08/01): Numerical Integration

W 12 (15/01): Dynamic Programming

W 13 (22/01): Application 1 - RBC model

W 14 (29/01): Application 2 - Inequality

W 15 (04/02): Review

Lecture Design

- We will use the 3,5 hours for lectures and discussion of exercises, in a flexible order ;)
- 10-min breaks after 50 minutes of teaching
- Bringing your laptop to class is encouraged to follow along with me coding is encouraged, but not mandatory!

Problem Sets

- Each lecture topic will be accompanied by a problem set containing mainly numerical exercises

- Working and submitting these problem sets is **optional**
- However, achieving at least 80% of the total score across all problem sets will grant you a "joker" for the exam
    - if you answer a exam question incorrectly or not at all, it will be disregarded and you get another question instead
- Information on how to submit problem sets will follow later

Material

- The Jupyter notebooks used in the lecture can be found on the LRZ Gitlab server and downloaded either directly or using Git (more on that later today):

  https://gitlab.lrz.de/ru67zef/cme-lmu

- The (partially) mandatory textbook for this class is **Mario J. Miranda and Paul L. Fackler (2004), *Applied Computational Economics and Finance*, MIT Press**

- Other textbooks that I will refer to as background reading (not required):
    - Kenneth Judd (1998), *Numerical Methods in Economics*, MIT Press
    - John Stachurski (2009), *Economic Dynamics - Theory and Computation*, MIT Press.
- Some sections will also have papers, as either required or recommended reading (compare syllabus).

Objectives

The goal of this course is to provide an introduction to computational tools for conducting numerical analysis in economics; hence,

- to provide you with a basic understanding of some fundamental techniques and algorithms used in numerical analysis and economic modeling
- to give you "hands-on" experience on how to implement these techniques in Python

"Secondary" goals are thus to

- get practice with Python
- more generally, get accustomed to coding and its principles

"Computers and mathematics are like beer and potato chips: two fine tastes that are best enjoyed together" (John Stachurski)

That said, it is important to note that

- this is neither a math class nor a CS course!
- the balance between hands-on coding and mathematical background may at times be tilted in the direction of the former!

Programming/Coding

Consider this course an opportunity to get familiar with programming in general and a specific language in particular.

- Indispensable skill in the context of economic modeling
- Programming has become increasingly important in all fields of economics
- Great skill to have for the (academic and non-academic) job market
- It is fun!

What is "Computational Economics" and why do we need it?

When do we use computers in economic analysis?

- Working with data
  - econometrics, machine learning
  - "black box" applications (e.g. STATA)
  - what happens in the box are implementations of numerical algorithms discussed in this course

- Illustration of theoretical "pen-and-paper" results
  - usually, the main result of a "deductive theory" paper is a general statement or closed-form expressions (*theorems*)
  - numerical analysis is used to illustrate this result for a given set of parameters
  - check for quantitative importance of (specific instances of) a result (e.g. the relative sizes of opposing effects)

Side note: in mathematics, a closed-form expression is a mathematical expression that can be evaluated in a finite number of operations (algebra and calculus)

- Solving models
  - in economics, a model can be solved analytically if a solution can be derived from a closed-form expression
  - many models (in particular dynamic ones) and hence many interesting problems lack a closed-form/analytical solution
  - therefore, computers have become an indispensable tool for conducting quantitative research in economics

Example

An agent maximizes her utility over two consumption goods. She has an initial endowment x for good 1. Moreover, she can convert good 1 in good 2, using a decreasing-returns-to-scale technology. Formally,

$$u(c_1, c_2) = \frac{c_1^{1-\nu}}{1-\nu} + \frac{c_2^{1-\nu}}{1-\nu}.$$

s.t.

$$c_2 = (x - c_1)^{\alpha}.$$

Taking first-order conditions and rearranging, we can find an optimality condition, that is, an expression in c_1:

$$(x - c_1)^{\alpha v - \alpha + 1} = \alpha c_1^{v}.$$

In the special case $v = 1$, it is easy to verify that

$$c_1^* = (1 + \alpha)^{-1} x$$

In the general case $v \neq 1$, we are not able to find a closed-form solution for c_1.

This indicates a *trade-off*: do we rely exclusively to analytical methods at the cost of limiting ourselves to special (and potentially not that relevant) cases? Or do we embrace numerical methods to be able to deal with more general problems?

Computational Economics - Defined

Wikipedia: "Computational economics is a research discipline at the interface of computer science, economics, and management science. This subject encompasses computational modeling of economic systems, whether agent-based, general-equilibrium, macroeconomic, or rational-expectations, computational econometrics and statistics, computational finance, computational tools for the design of automated internet markets, [...]. Some of these areas are unique to computational economics, while others extend traditional areas of economics by solving problems that are difficult to study without the use of computers and associated numerical methods."

More narrow: "Computational economics uses computer-based economic modeling for the solution of analytically and statistically formulated economic problems."

Deductive Theory vs. Computational Methods

Compare Judd (1997)

What is a theory?

- collection of concepts, definitions and assumptions
- a *theoretical analysis* determines the implications of a theory

Deductive (conventional) theory...

- is concerned with "proving theorems" (or lemmas, or propositions), i.e. "pen-and-paper results"
- can determine the "qualitative structure of a model"
- can rarely answer all important questions; if the theorist is unable to prove general results, she needs to turn to special cases which are tractable

Drawbacks of deductive theory:

- special cases may sacrifice model elements of first-order importance
- nonquantitative, i.e. no indication about the relevance of a result

Deductive theory is one approach to theoretical analysis, but is not a synonym.

Distinguish between "mathematical economics and economic theory".

Computation is an alternative way to conduct theoretical analysis.

Consider Hamming's motto: *The Goal of Computing is Insight, Not Numbers*.

Arguments against "computational theory":

- Computer programs as a ``black box''
  - Easy remedy: better exposition
  - Einstein: a model should be ``as simple as possible, but not simpler''
- Numerical analysis contains approximation errors
  - "in economic theory, [...] the issue is not whether we use approximation methods, but where in our analysis we make approximations, what kind of approximation errors we tolerate and which ones we avoid, and how we interpret the inevitable approximation errors." (Judd 1997, p. 918)
  - simple cases in deductive analysis also make approximation errors
  - of course, even very complex computer models are simplifications of the real world
- Solving a model numerically also considers a special case, e.g. a given calibration
  - importance of sensitivity analysis/robustness checks
  - computation allows the modeler to consider a (finite) set of specific instances of a theory, not just one special case

In a nutshell: "[S]ince any theoretical analysis is really an approximation of the economic reality we are trying to model, we have to ask which is more important: the development of theories simple enough for theorem-proving, or examining more reasonable theories using less precise numerical methods." (Judd 1997, p. 909)

At the end of the day, computational methods are another useful instrument in your economist's "toolbox".

As with any tool, they will not be applicable to *every* problem, but may be the best (and sometimes the only) way forward for *some* problems.

If all you have is a hammer, everything looks like a nail.

Rules of Thumb for Doing Good Computational Work in Economics (via Tony Smith)

- Start with the simplest possible model, preferably one with an analytical solution.
- Add features incrementally.
- Never add another feature until you are confident of your current results.

- Use the simplest possible methods. Use one-dimensional algorithms as much as possible.
- Accuracy is more important than speed or elegance.
- Use methods that are as transparent as possible (i.e., methods for which the computer code reflects as closely as possible the economic structure of the problem).

- When you learn (or develop) a new method, test it on the simplest possible problem, preferably one with an analytical solution.
- Dan Bernhardt's rule: If you have n errors in a piece of code and you remove one, you still have n errors. Scrutinize your results, even if they look right. Look for anomalies. *Assume your code is wrong until proven otherwise.*
- Graph, graph, graph. Two-dimensional graphs are more informative than three-dimensional graphs.

- Be able to replicate all of your intermediate and final results instantly. Save exact copies of the code used for each run, together with inputs and outputs (*version control*!).
- Watch the computations as they proceed.
- Look for hidden structure. Always compute (and print out) a few more numbers than you need.

Python: An Overview

python

In this course, we're going to use exclusively **Python** the lectures and tutorials; in other words, examples, exercises, solutions etc. will be provided only in Python.

You are required to work on the exam question in Python as well.

Wikipedia: "Python is a widely used **high-level programming language** for **general-purpose** programming, created by Guido van Rossum and first released in 1991.

An **interpreted language**, Python has a design philosophy which emphasizes **code readability** (notably using **whitespace indentation** to delimit code blocks rather than curly braces or keywords), and a syntax which allows programmers to **express concepts in fewer lines of code** than possible in languages such as C++ or Java. The language provides constructs intended to enable writing clear programs on both a small and large scale."

Fun fact: Python is not named after the animal, but after *Monty Python's Flying Circus*.

https://www.youtube.com/watch?v=FQ5YU_spBw0

```
In [3]: display.Image('monty.jpg', width = 400)
```

```
Out[3]:
```

Some characteristics

- High-level programming language
  - emphasis on readability
  - easy to learn and easy to use

"I have this hope that there is a better way. Higher-level tools that actually let you see the structure of the software more clearly will be of tremendous value." (Guido van Rossum)

- Interpreted / dynamically typed
  - no explicit compilation step (as e.g. in C++ or Fortran)
  - inefficient memory access
  - As a result, running an application in Python is (typically) slower than in lower-level, compiled languages

"Given this inherent inefficiency, why would we even think about using Python? Well, it comes down to this: Dynamic typing makes Python easier to use than C. It's extremely flexible and forgiving, this flexibility leads to efficient use of development time, and on those occasions that you really need the optimization of C or Fortran, Python offers easy hooks into compiled libraries.

It's why Python use within many scientific communities has been continually growing. With all that put together, **Python ends up being an extremely efficient language for the overall task of doing science with code**." (https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/)

Keep it in mind: *programming time is the sum of coding time and running time!*

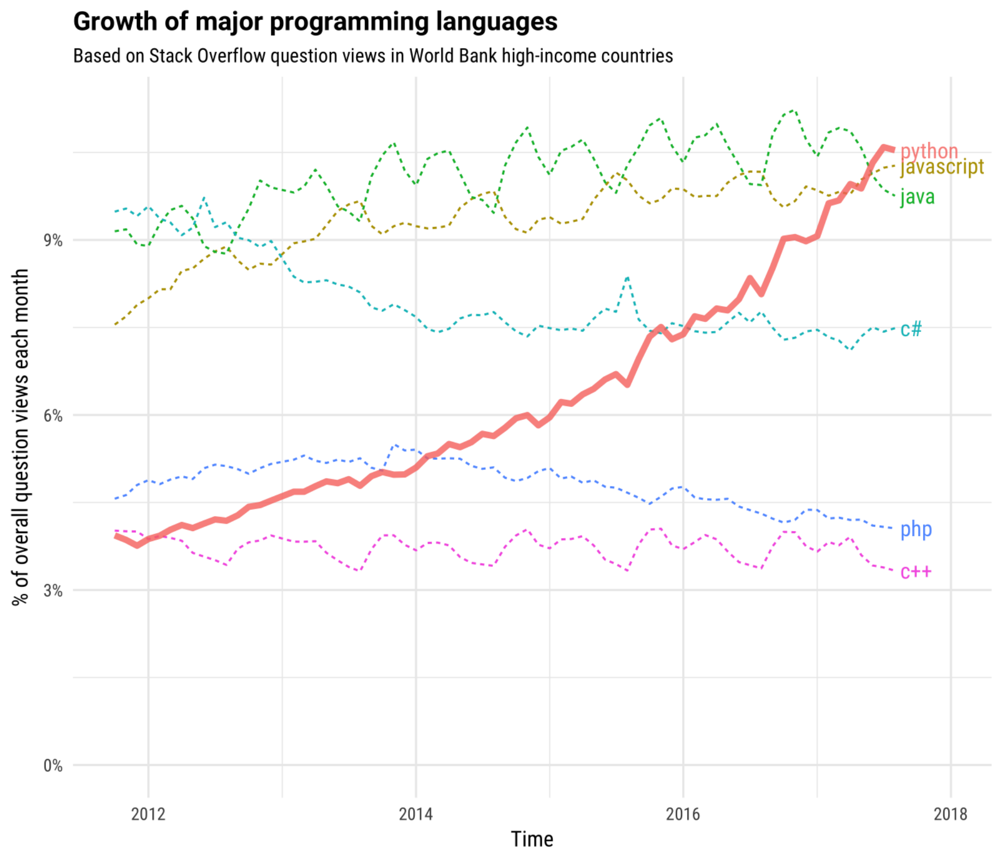
- Free and open-source
  - large community of users and developers ("Pythonistas"); many applications have been implemented in packages which are easy to downloaded and import
  - extensive documentation

- General-purpose language; examples for applications include:
  - "Scientific Computing" (**Numpy/Scipy**, **Matplotlib**: matrix operations, optimization, root finding, plotting, etc.)

- Working with data (**Pandas**, **Statsmodel**)
- Parsing text files
- Web scraping (**Beautiful Soup**, **scrapy**)
- Machine Learning (**Scikit-Learn**)
- Games
- Creation of GUIs, web services etc.

In [4]:
```
display.Image('growth_major_languages.png', width = 500, height = 400)
```

Out[4]:



Source: https://stackoverflow.blog/2017/09/06/incredible-growth-python/

Why Should You Learn Python?

Easy Answer: because it is required for the course! :)

General answer: "trifecta of greatness"

- ease of use
- open source with a large dev community
- general purpose

If you have no or very little experience with programming and scripting languages:

- programming skills are highly useful and valuable both on the academic and non-academic job market
- due to its emphasis on readability and simple syntax, Python is (most likely) the easiest way to get into programming while still being a powerful language

Why should you learn Python *in addition* or *instead of* another programming language?

General point 1: having more tools at your disposal is never a bad idea!

If all you have is a hammer, everything looks like a nail.

General point 2: there is no such thing as the "best programming language"

- what a "right" language is depends very much on the task(s) at hand

"To be honest, I really hate those types of questions: *Which _ is the best?* (_ insert *programming language, text editor, IDE, operating system, computer manufacturer* here). This is really a nonsense question and discussion. Sometimes it can be fun and entertaining though, but I recommend saving this question for our occasional after-work beer or coffee with friends and colleagues." (https://sebastianraschka.com/blog/2015/why-python.html)

General point 3: if you are experienced in another programming language, the cost of learning Python is probably small

"Just think about, you learned how to swing a hammer to drive the nail in, how hard can it possibly be to pick up a hammer from a different manufacturer?" (https://sebastianraschka.com/blog/2015/why-python.html)

Python vs. MATLAB

- Everything that you can do in (basic) MATLAB (scientific computing, graphics) you can also do in Python (using packages like Numpy, Scipy, Matplotlib), with little adjustment cost
- Not everything that you can do in Python you can do in MATLAB (e.g. data analysis)
- Data Science/Big Data Analysis: Python and R seem to be the most popular languages by a mile
- No general speed advantages: http://julialang.org/benchmarks/
- "Knockout blow": MATLAB is rather pricey, Python is open source
- More arguments: https://folk.uio.no/jeanra/Informatics/UsePythonNotMatlab.html

Main advantages of MATLAB:

- Quality of (some) algorithms and routines
- Slightly easier syntax than Python/Numpy
- "Networking" aspect: (still) predominant language in computational economics

Python vs. R

Disclaimer: I have no experience with R, so all of the following is based on Google and hearsay :)

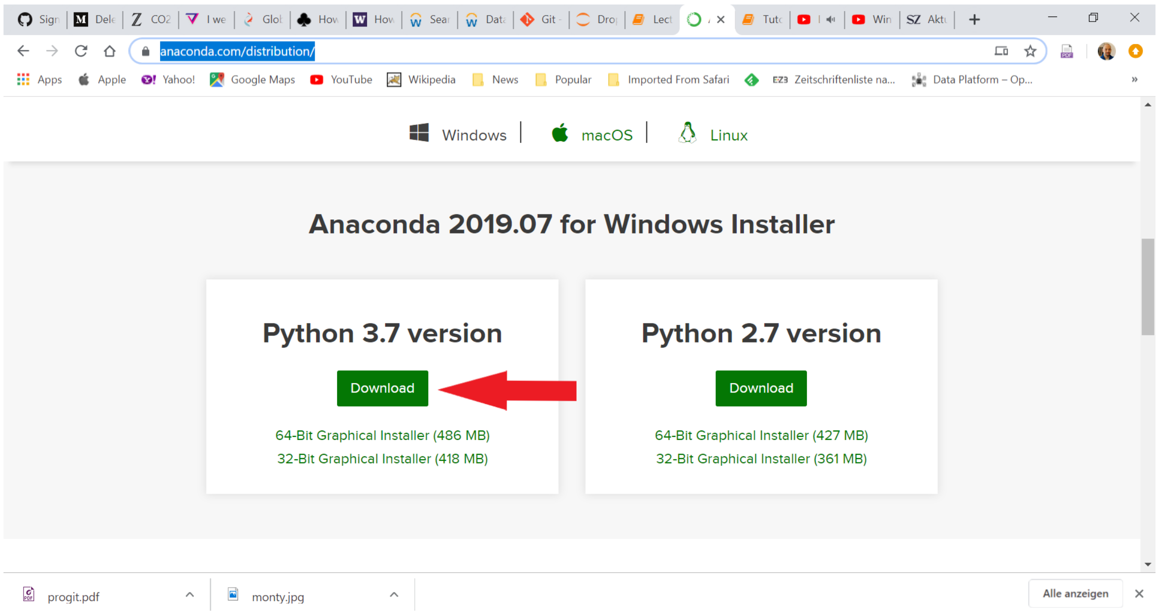
- R has a more narrow application spectrum: data science/data analysis $\Rightarrow$ not general purpose
- In the realm of data science/data analysis, both are very close substitutes
- If you don't know either, Python seems to have considerably lower learning cost

Installing Python

- There exist different Python *distributions*; your computer may already have a Python version installed
- For this course, I recommend to use the **Anaconda** distribution
  - download for free at https://www.anaconda.com/distribution/
  - easy to install, manage and update
  - contains all relevant scientific packages (Numpy, Scipy, Pandas, etc.)

In [5]:
```
display.Image('anaconda.png', width = 800)
```

Out[5]:



Anaconda 2019.07 for Windows Installer

Python 3.7 version	Python 2.7 version
64-Bit Graphical Installer (486 MB)	64-Bit Graphical Installer (427 MB)
32-Bit Graphical Installer (418 MB)	32-Bit Graphical Installer (361 MB)

After downloading for Windows or Mac, install Anaconda3 by following the instructions.

Important points:

- Install the latest version.
- If you are asked during the installation process whether you'd like to make Anaconda your default Python installation, say yes
- Otherwise, you can accept all of the defaults

Python 2 or Python 3?

- "Python 2.x is legacy, Python 3.x is the present and future of the language"
  - Python 2.7 is no longer actively developed, but still widely used; some packages are written exclusively for Python 2
  - There are some differences with respect to the syntax
  - When writing new code, it is relatively easy to write it in a way so that it runs under both Python 2 and Python 3

- For this course, it doesn't matter too much which one we use; since **Python 3.7** is the "current" version, we go this way
- With Anaconda, it is fairly straightforward to have an environment for each of them, and switch between them as need be (see https://conda.io/docs/py2or3.html)

For more information, compare for example

- https://wiki.python.org/moin/Python2orPython3
- http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html

How to Use Python

- command line (**Ipython**)
- scripts; make sure to get a "good" text editor (e.g. Sublime Text 2/3, Notepad++, etc.)!
- **Spyder** (Scientific Python Development Environment)
  - Spyder is included in Anaconda

Working with Jupyter Notebooks

This environment is a *Jupyter notebook* (*Ju*lia, *Pyt*hon and R). It basically provides a *browser-based* interface to Python, allowing you to run all your Python code and get the output from your code in a browser window (Google Chrome, Mozilla Firefox, etc.).

A very recent evolution of Jupyter notebooks is *Jupyter lab*: it also runs on your browser, but has a somewhat different interface.

What's more, you can add formatted text cells like this one, and even include mathematical formulas (as we have already seen above), based on the Latex syntax.

In addition, you can also incorporate images. This not only makes a notebook a great tool for *writing and documenting code*, it is also great for teaching and sharing research. That's why we will make heavy use of Jupyter notebooks in this course.

Note that all course materials - lecture notes, exercises, solutions - will be provided as Jupyter notebooks.

For the exam and the (optional) problem sets, you will be asked to submit Juypters notebook as well!

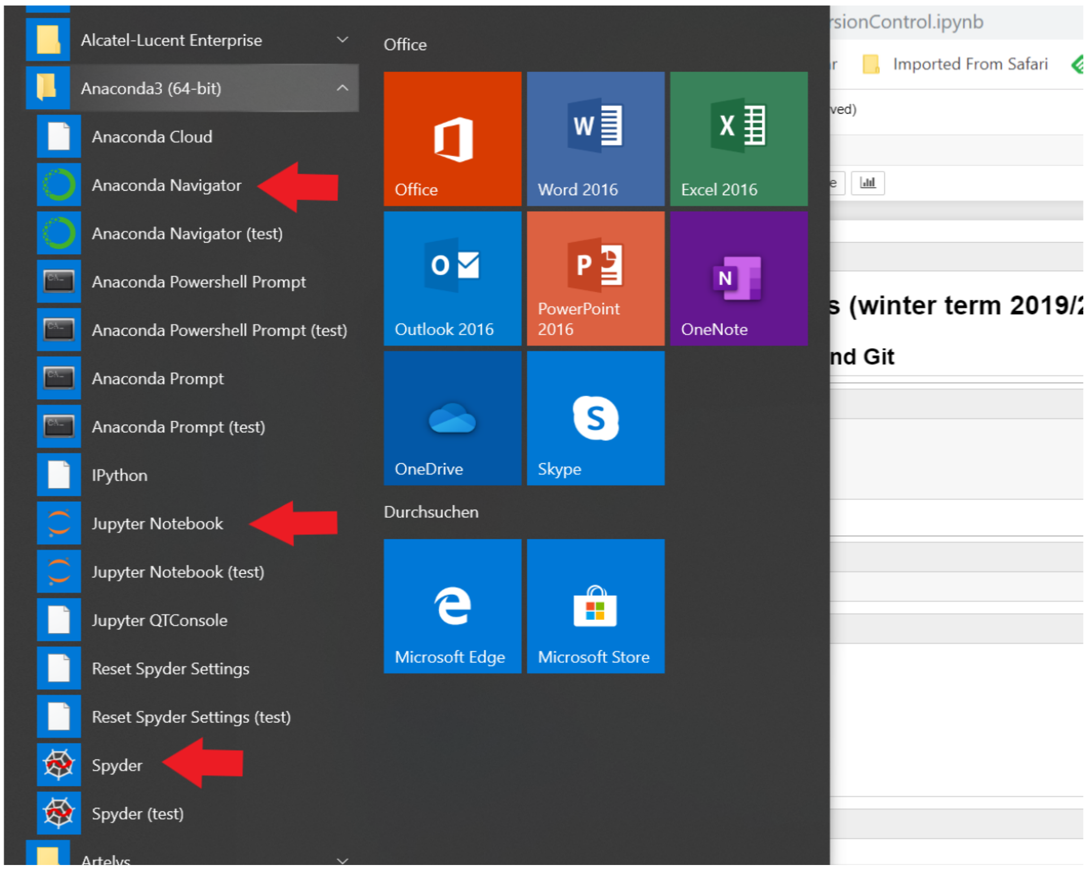
To start a Jupyter notebook on Windows: click on *Start* -> *Anaconda3*, and

- use Jupyter Notebook tab
- use the Anaconda Navigator
- use the Anaconda Powershell Prompt and type **Jupyter notebook** (**Jupyter lab**)

You then see the Jupyter dashboard and can navigate to your working directory. If there is a Jupyter notebook in your directory (with suffix **.ipynb**), you can open it by clicking on it. Otherwise, you can open an empty notebook by clicking on *New* at top right and select Python 3.

```
In [6]: display.Image('start_jn.png', width = 800)
```

```
Out[6]:
```



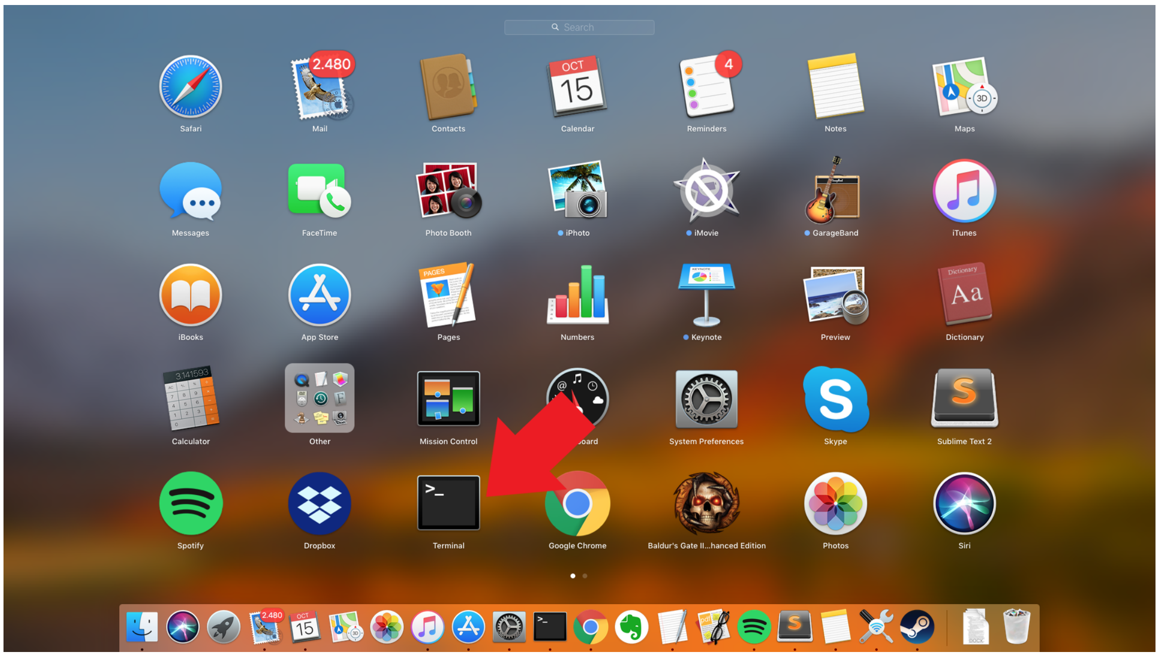
To start a Jupyter notebook on Mac:

- use the Anaconda navigator (via the Launchpad -> *doesn't seem to always work*)
- open a terminal window and type **Jupyter notebook** (**Jupyter lab**).

You then see the Jupyter dashboard and can navigate to your working directory. If there is a Jupyter notebook in your directory (with suffix **.ipynb**), you can open it by clicking on it. Otherwise, you can open an empty notebook by clicking on *New* at top right and select Python 3.

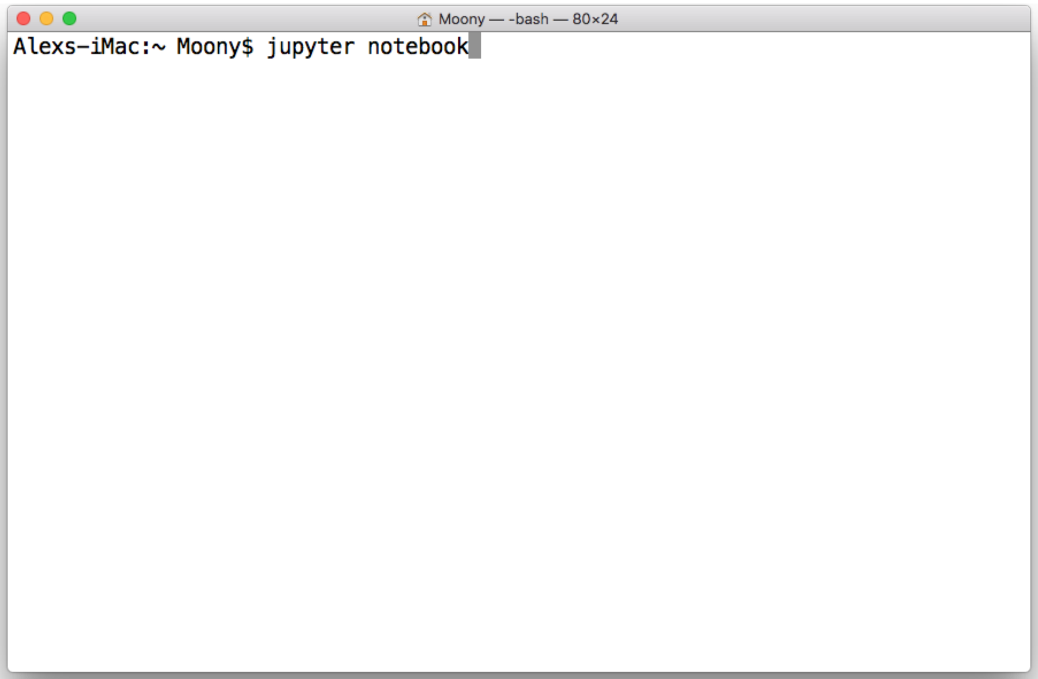
```
In [7]: display.Image('mac_terminal.png', width = 800)
```

Out[7]:



```
In [8]: display.Image('mac_terminal2.png', width = 800)
```

Out[8]:



```
Alexs-iMac:~ Moony$ jupyter notebook
```

A Jupyter notebook has two types of cells. A new cell is by a default a *code cell*. In contrast to a *text cell*, it has some blue writing ("**In [ ]**:") to the left of it. Once you run it, a number appears in the bracket. You can run any cell by either clicking the **Run** button in the toolbar above or by pressing **Shift + Enter**.

Running a cell processes its output: for a text cell, it just formats the text. For a code cell, it executes the code and prints the output below. Jupyter then either creates a new code cell below or (if already there) jumps to the next cell.

```
In [9]: print("This is a code cell.")
```

```
This is a code cell.
```

A code cell can be made into a *text cell* like this one by choosing "**Markdown**" in the drop-down menu above in the toolbar, or by selecting a cell, pressing the **Esc** key, followed by **m** (*Markdown* is a type of text format).

The previous statement was an example of Jupyter notebook's *modal editing system*. This means that the effect of typing at the keyboard depends on which mode you are in.

The two modes are

- *Edit mode* (indicated by a green border around one cell): whatever you type appears as is in that cell
- *Command mode* (indicated by a blue border): key strokes are interpreted as commands

To switch from edit mode to command mode, hit **Esc** or **Ctrl-M** (**Strg-M** on a German keyboard).

From command mode to edit mode, hit **Enter** or click in a cell.

The modal behavior of the Jupyter notebook is a little tricky at first but very efficient when you get used to it.

More information on Jupyter notebooks and the Anaconda environment, including a couple of tricks to optimize your workflow (e.g. tab completions) can be found here:

https://lectures.quantecon.org/py/getting_started.html#jupyter-notebooks

Moreover, once you have installed Anaconda on your computer, open a notebook and try out the different features in the menu and tool bars, to familarize yourself with the environment!

Finally, note that you can save Juypter notebooks in other formats, for example in HTML or as a pdf, in case you want to print them, share them etc.

To do so, click on **File** in the menu bar and then choose **Download as** plus the desired format.

"Homework": Please try to install Anaconda3 and open a Jupyter notebook until next week's session! If you run into problems, let me know!