



# Introduction

## Mini-Course: Heterogenous Agent Macro

---

Jeppe Druedahl

2025



1. Introduction

2. Programming in Python

# Introduction

---

- **Central economic topics:**

1. Consumption-saving with risk and constraints (single-agent)
2. Heterogeneous agents in general equilibrium models
  - 2.1 Long-run effects on aggregate outcomes
  - 2.2 Short-run effects on aggregate outcomes
  - 2.3 Drivers of inequality

- **History:**

1. Heathcote et al. (2009), »Quantitative Macroeconomics with Heterogeneous Households«
2. Kaplan and Violante (2018), »Microeconomic Heterogeneity and Macroeconomic Shocks«
3. Cherrier et al. (2023), »Household Heterogeneity in Macroeconomic Models: A Historical Perspective«
4. Auclert et. al. (2025), »Fiscal and Monetary Policy with Heterogeneous Agents«

- **Central technical method:** *Programming in Python*

# Macroeconomic Models with Heterogeneous Agents

- **Model components:**

1. Optimizing individual agents (households + firms)
2. Idiosyncratic and aggregate risk
3. Information flows (who knows what when  $\Rightarrow$  often everything)
4. Market clearing (Walras vs. search-and-match)

- **Insurance/markets:**

*Complete*  $\rightarrow$  idiosyncratic risk insured away  $\sim$  representative agent

*Incomplete*  $\rightarrow$  agents need to *self-insure*

- **Heterogeneity:**

*Ex ante* in preferences, abilities etc.

*Ex post* after realization of idiosyncratic shocks

- **HANC:** Heterogeneous Agent *Neo-Classical* model  
(Aiyagari-Bewley-Hugget-Imrohoroglu or Standard Incomplete Markets model)
- **HANK:** Heterogeneous Agent *New Keynesian* model  
(i.e. include price and wage setting frictions)

- **Topics:**

1. Consumption-saving
2. Stationary equilibrium
3. Transitional dynamics
4. HANK models

- **Teaching philosophy:**

1. Go in depth - *from theory to implementation*
2. Not a literature review - *key entrances to literature*
3. Focus on tools - *easier to read economic insights*

1. **Assumed knowledge:** Similar to my undergraduate course **Introduction to Programming and Numerical Analysis**  
Content: Python + VSCode + git  
Preparation: **Video playlist** (~10 hours at normal speed)
2. **Updated Python:** Install (or re-install) newest Anaconda
3. **Packages:** `pip install quantecon, EconModel, consav`
4. **GEModel tools:**
  - 4.1 Clone the GEModelTools repository
  - 4.2 Locate repository in command prompt
  - 4.3 Run `pip install -e .`
5. **Other repositories used in the course:**
  - **EconModelNotebooks**
  - **ConsumptionSavingNotebooks**
  - **GEModelToolsNotebooks**
  - **MiniCourse-HetAMacro**

# Programming in Python

---



# References (pointers)

- **Variables are references to an instance of an object**
- A **class** defines the **type** of an object
  - .attribute, state
  - .method(), action (incl. changing self)
- Inheritance (of methods) (class Child(Parent))
- Arithmetic operators (e.g. +, \*, /, //, \*\*, %) combine objects
- **= assigns a reference** (*not a copy!*)

**Question:** What does a end up as? What if a = [1,2,3]?

```
1 a = np.array([1,2,3])
2 b = a
3 c = a[1:] # slicing
4 b[0] = 3 # indexing
5 c[0] = 3
```

# Types and in-place operations

- **Atomic types:** int, float, str, bool, etc.
- **Containers** list, tuple, dict, set, np.array, etc.
- **Mutables** (e.g. list, np.array) can change in-place
  1. **In-place operators** (`+=`, `-=` etc.)
  2. **Slicing:** `x[:] = x + y`
- **Immutable** (e.g. atomic types and tuples) can never change

**Questions:** What does y end up as?

```
1 x = np.array([1,2,3])
2 y = x
3 x += 1
4 x[:] = x + 1
5 x = x + 1
```

# Functions and scope

- **Functions are objects** (can e.g. be arguments in functions)

Unlike in math:

1. Can change its arguments (side-effects)
  2. Can call itself (recursion)
- Decorators change function behavior (e.g. @numba.njit)
  - Variables can both be **local scope** (good) or **global scope** (bad)

**Questions:** What is the output?

```
1 a = 1
2 def f(x):
3     return x+a
4 print(f(1))
5 a = 2
6 print(f(1))
```

# Computational tree and branches

- **Comparison** (`==`, `!=`, `<`, `<=`, `not`, `and`, or etc.)
- **Conditionals** (`if`, `elif`, `else`)
- **Loops** (`for`, `while`, `continue`, `break`)
- **Convergence** (tolerance in optimizer or root-finder/equation-solver)

**Questions:** How could this be implemented with a while loop?

```
1 x = x0
2 for i in range(n):
3     y = evaluate(x)
4     if check(y): break
5     x = update(x,y)
6 else:
7     raise ValueError('did not converge')
```

# Decimal numbers are not exact

- **Never use exactness for decimal numbers**
  - Order of computation matter
  - Best with numbers are around 1 (underflow and overflow)
- Division, exp, log etc. are (costly) approximations
- **Function approximation and interpolation often needed**

**Questions:** Which are True and which are False?

```
1 print(0.1 + 0.2 == 0.3)
2 print(0.5 + 0.5 == 1.0)
3 print(np.isclose(0.1+0.2,0.3))
4 print(np.isclose(1e-200*1e200*1e200*1e-200,1.0))
5 print(np.isinf(1e-200*(1e200*1e200)*1e-200))
6 print(np.isclose(1e200*(1e-200*1e-200)*1e200,0.0))
```

# Pseudo random numbers

- **Only one seed** (randomness not assured across seeds)
- State of random number generator can be reset
- **Monte Carlo** simulation and integration
  1. Static alternative: Use **quadrature rules**
  2. Dynamic alternative: Discretize and derive **transition matrix**

**Questions:** What is  $z$  equal to?

```
1 rng = np.random.default_rng(123)
2 s = rng.bit_generator.state
3 x = rng.normal(size=5)
4 y = rng.normal(size=5)
5 rng.bit_generator.state = s
6 z = rng.normal(size=5)
```

# Documentation and debugging

- **No code is self-explanatory** (for others, incl. future you)
- **Write documentation** (use *github-copilot*)
  1. The comments explain humans what the code does.
  2. The code makes the computer do what the comments say
- Important **design patterns**:
  1. Use namespaces (be aware of scope) and meaningful names
  2. No repetition of code-lines  $\Rightarrow$  single-purpose functions/methods
  3. Use assert (also print and plot intermediate results)
  4. Use try-except
- **Run from top to bottom** (make shortcut)

Replication: [datacodestandard.org](https://datacodestandard.org)
- **Debugging** (see 02. Debugging.ipynb)
  1. Errors are (almost) always simple
  2. Go through code step-by-step (*manually* or *debugger*)

- **High level languages:**

1. **MATLAB:** Costly and not better.
2. **R:** Better at statistics and data work, but not pure numerical work.
3. **Julia:** Faster than Python (incl. numba), slower than C++.  
Smallish community.

- **Low level languages:**

1. **C++:** State-of-the-art for fastest code.
2. **Fortran:** No benefits relative to C++ (only legacy...).

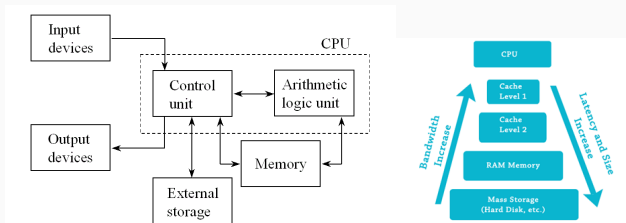
- **Hardware:**

1. CPU: Fastest cores.
2. GPU: Many more cores, but specialized for e.g. linear algebra.
3. TPU (Google): Even more specialized at AI (machine learning)



# CPUs are complex

- **Instruction set** (assembly) is not just add, subtract, etc.
  1. Work on vectors (SIMD)  $\Rightarrow$  *homogeneity is good*
  2. Out-of-order execution  $\Rightarrow$  *predictability is good*
  3. Caching  $\Rightarrow$  *latest read memory can be accessed quickly*



- **Compilers can optimize a lot**  $\Rightarrow$  use existing libraries
- **Parallelisation:** *Start up costs*
  1. **Hardware:** Cores vs. CPUs vs. sockets vs. computers
  2. **Software:** Shared memory (e.g. OpenMP) or not (e.g. MPI)

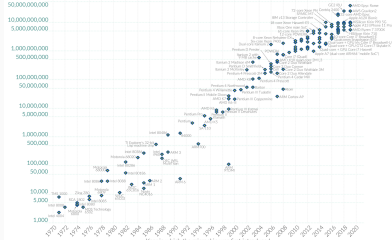
# Moore vs. Amdahl

**Moore's Law:** The number of transistors on microchips doubles every two years.

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

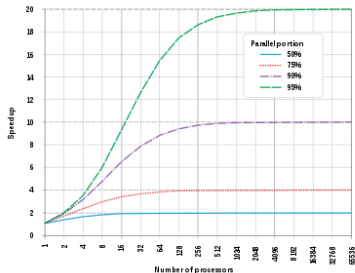
Our World in Data

Transistor count



Data source: <https://www.intel.com/content/www/us/en/developer/articles/technology-blogs/2017/09/27/transistor-count.html>  
OurWorldInData - Research and data to make progress against the world's biggest problems.  
Licensed under CC-BY by the authors Hans Rosling and Max Roser.

Amdahl's Law



## 1. **Moore's law:** Exponential growth in computational power

1.1 Originally: Faster cores (calculations per time unit)

1.2 Now: More cores per CPU / per computer / per dollar

## 2. **Amdahl's law:** Sequential code becomes the bottleneck

2.1 Time in tasks done in parallel  $\rightarrow 0$

2.2 95% done in parallel  $\rightarrow$  max 20x speed-up

## Need for speed (03. NeedForSpeed.ipynb)

- **Computation time vs programmer time**

Use not-too-model-specific insights  $\Rightarrow$  *better algorithm*

- **Premature optimization is the root of all evil!**

Use *line-profiler*!

1. Use available code: Stand on the shoulder of giants
2. In numpy: Use vectorization
3. Else: Use numba

- **Automatic differentiation?** Use JAX (or PyTorch)

- **Faster still?** Implement bottleneck in C++ and call from Python

# EconModelClass

- **Package:** EconModel
- **Purpose:**
  1. Make it easy to write well-structured code.
  2. Provide standard functionality for copying, saving and loading.
  3. Provide an easy interface to call numba JIT compiled functions.
  4. Provide an easy interface to call C++ functions.
- **Notebooks:**

EconModelNotebooks\01. Using the EconModelClass.ipynb  
(not the C++ part)
- **Video:** [Youtube - EconModel](#)