Programming Experience

Engineers or Scientists?

- Solve the problem
- Programming is our tool (easy to learn)
- Accuracy comes first (hard to code)
- Efficiency comes next (even harder)
- Reproducibility is a must

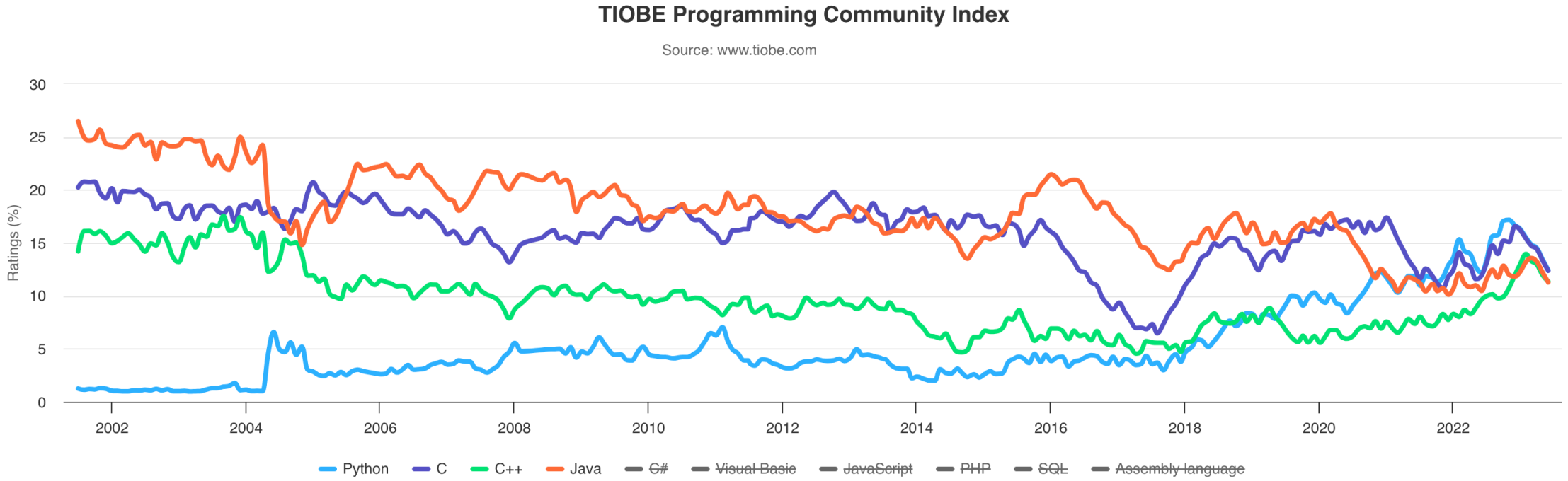
Programming Environment

- **Basic Linux Packages**: gcc, g++, gfortran, icc, ifort, nvcc, wget, git, screen, sshfs
- **Git**: GitHub and Gitlab are two popular websites for code hosting.
- **IDE**: VSCode is popular for Python, C/C++, Fortran, etc. Its extension is extremely powerful
  - Python
  - C/C++
  - Fortran
  - Remote - SSH
  - Jupyter
  - Markdown

Programming Language

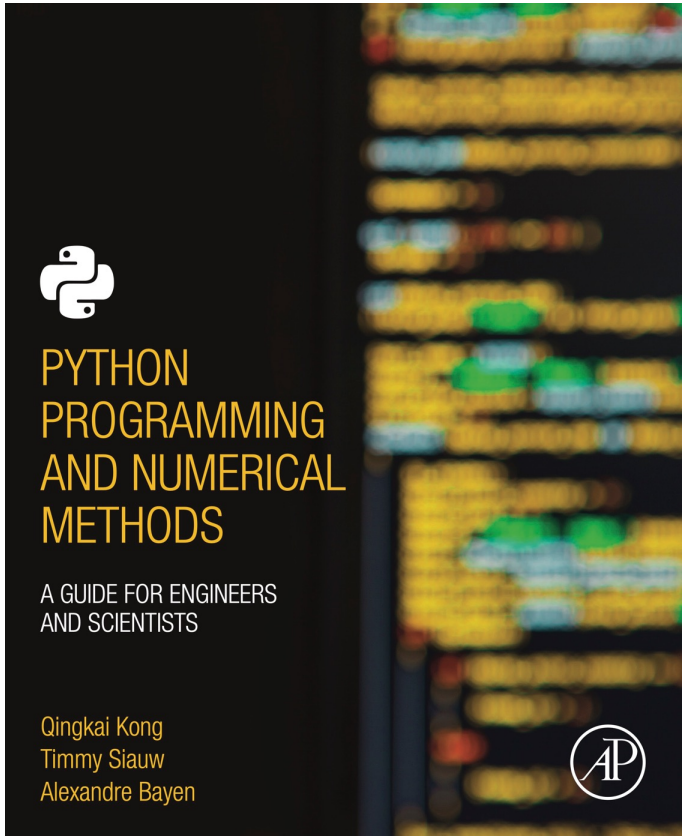
- **Compiled Language**: C/C++, Fortran, …
- **Interpreted Language**: Python, Julia, …

Jun 2023	Jun 2022	Change	Programming Language
1	1		Python
2	2		C
3	4	▲	C++
4	3	▼	Java



Programming Language

- **Python** is a good choice (community)
  - Python 3.14 will be faster than C++
- **Anaconda**: package management and deployment
- **Jupyter Notebook**: code, equations, visualizations and text.
- **Prevailing working mode**:
  - VSCode + Jupyter Notebook



https://pythonnumericalmethods.berkeley.edu/notebooks/Index.html

Computing in Python

- Cache
  - the fast axis of the array is the last axis

```python
import numpy as np

# create a 2D array
a = np.random.rand(1000, 1000)

# loop over the array
for i in range(1000):
    for j in range(1000):
        a[i, j] += 1.0

# bad example!
for i in range(1000):
    for j in range(1000):
        a[j, i] += 1.0
```

Computing in Python

- Cache
  - the fast axis of the array is the last axis
- Vectorization

```python
import numpy as np

# create two arrays
a = np.random.rand(1000, 1000)
b = np.random.rand(1000, 1000)

# calculate the sum of two arrays by
c = np.zeros((1000, 1000))
for i in range(1000):
    for j in range(1000):
        c[i, j] = a[i, j] + b[i, j]


# calculate the sum of two arrays by
c = a + b

# or equivalently
c[:] = a[:] + b[:]
```

Computing in Python

- Cache
  - the fast axis of the array is the last axis
- Vectorization
- Numpy Functions
  - based on BLAS and LAPACK

```
1  import numpy as np
2
3  # create two arrays
4  a = np.random.rand(1000, 1000)
5  b = np.random.rand(1000, 1000)
6
7  # calculate the sum of two arrays
8  c = np.add(a, b)
```

JIT Computing in Python

- Just-in-time Compilation
  - Game changer!
  - Always JIT your code
  - That's why Julia is fast

```python
import numpy as np
from numba import jit

# define a function
@jit(nopython=True)
def myadd(a, b):
    c = np.zeros((1000, 1000))
    for i in range(1000):
        for j in range(1000):
            c[i, j] = a[i, j] + b[i, j]
    return c

# create two arrays
a = np.random.rand(1000, 1000)
b = np.random.rand(1000, 1000)

# calculate the sum of two arrays by using
c = myadd(a, b)
```

Parallel Computing in Python

- **Hardware**: Memory, Cache, CPU, GPU, TPU, …
- **Terminology**: Thread, Process, Core, Node, Cluster, Supercomputer, Cloud
- **GIL**: The parallelism of Python is limited by the global interpreter lock
- **No real parallel in Python**

Parallel Computing in Python

- **Thread-level**
  - **Multiprocessing:** similar to OpenMP
  - **Multiprocess**: uses dill instead of pickle for serialization
- **Process-level**
  - **MPI4py:** self-explanatory
  - **Dask:** support distributed computation
- **GPU-level**
  - **Numba:** JIT compiler that supports GPU computing
  - **Cupy:** open-source array library for GPU-accelerated computing with Python
- **IO is bottleneck**
  - please be aware of

OOP is All You Need

- **Public, Protected, and Private Attributes**
- **Encapsulation**, **inheritance**, & **polymorphism**

```
class Receiver:
    def __init__(self, location_x, component):
        self.location_x = location_x
        self.component = component

    def printInfo(self):
        print('name: %s, age: %d' % (self.location_x, self.component))
```

OOP is All You Need

- Linear Inversion $d = Fm$
- Nonlinear Inversion $d = F(m)$

Level of abstraction

- Vector
- Operator
- Solver
- Problem
- Workflow

OOP is All You Need

- **Vector**: hypercube for model and data

```
import sys

class myVector:
    """An abstract vector class"""
    def __init__(self):
        pass;
    def die(self,cls):
        """ Helper function to exit when class in not defined"""
        print("Method ",cls," has not been overritten")
        self.exit(-1)
    def add(self,vec):
        """Add the contents of another vector to the current vector"""
        self.die("add")
    def scale(self,scalar):
        """Scale a vector by a scalar"""
        self.die("scale")
    def clone(self):
        """Make a copy of the vector"""
        self.die("clone")
    def dot(self,vec):
        """Dot product with another vector"""
        self.die("dot")
    def random(self):
        """Fill vector with random numbers"""
        self.die("random")
    def getNdArray(self):
        """Return a numpy array version of the vector"""
        self.die("getNdArray")

    # methods that are not provided but are nice to have for later use
    def checkSame(self,vec):
        """Check to see if two vectors are the same size"""
        self.die("checkSame")
    def zero(self):
        """Set all elements to zero"""
        self.die("zero")
```

OOP is All You Need

- **Vector**: hypercube for model and data
- **Operator**: forward, adjoint, dot-product

```
import math
class myOperator:
    """Generic operator class"""
    def __init__(self):
        pass;

    def setDomainRange(self,domainV,rangeV):
        """Set the domain and range vectors"""
        self.domainV=domainV.clone()
        self.rangeV=rangeV.clone()

    def checkDomainRange(self,mod,dat):
        """Check to make sure spaces match mod->domain dat->range"""
        if not self.domainV.checkSame(mod):
            raise Exception("Domain does not match")
        if not self.rangeV.checkSame(dat):
            raise Exception("Range does not match")

    def forward(self,add,model,data):
        raise Exception("Must override forward")

    def adjoint(self,add,model,data):
        raise Exception("Must override adjoint")

    def dotProduct(self):
        x=self.domainV.clone()
        x2=self.domainV.clone()
        y=self.rangeV.clone()
        y2=self.rangeV.clone()
        x.random()
        y.random()
        self.forward(False,x,y2)
        self.adjoint(False,x2,y)
        erro = abs(x.dot(x2) -y.dot(y2))
        if(erro >1e-4):
            raise Exception("Does not pass product test")
        else:
            print('Absolute error = ',erro)
            print("Passed dot product test!")
```

OOP is All You Need

- **Vector**: hypercube for model and data
- **Operator**: forward, adjoint, dot-product
- **Solver**: gradient-based methods

1. Preconditioned Steepest Descent: PSTD
2. Preconditioned Nonlinear Conjugate Gradient: PNLCG
3. Quasi-Newton l-BFGS method: LBFGS
4. Quasi-Newton Preconditioned l-BFGS method: PLBFGS
5. Truncated Newton method: TRN
6. Preconditioned Truncated Newton method: PTRN

My Python implementation of SEISCOPE optimization toolbox: reverse communication solver

https://github.com/Haipeng-ustc/SWIT-1.0/tree/main/dev/toolbox-dev/optimization

Scipy.optimize.minize

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

OOP is All You Need

- **Vector**: hypercube for model and data
- **Operator**: forward, adjoint, dot-product
- **Solver**: gradient-based methods
- **Problem**: the combination of the operator and solver

$$\chi(\mathbf{m}) = \frac{1}{2}\| f \cdot M \cdot W(\mathbf{d}_{\text{obs}} - R \cdot F(\mathbf{m}))|_2^2$$

$$\nabla_{\mathbf{m}}\chi = -\frac{\partial F^T}{\partial \mathbf{m}} W^T \cdot M^T \cdot f^T \cdot R^T(\mathbf{d}_{\text{obs}} - R \cdot F(\mathbf{m}))$$

OOP is All You Need

- **Vector**: hypercube for model and data
- **Operator**: forward, adjoint, dot-product
- **Solver**: gradient-based methods
- **Problem**: the combination of the operator and solver
- **Workflow**: i.e., Traveltime Inversion + FWI + RTM

Unit Test

Items	How to know the code is correct?
Forward propagator	Benchmark with well-built propagators
Adjoint propagator	Pass the **dot-product** test
Gradient	Build the Jacobian explicitly by point-wise parameter perturbation
Solver	Test on rosenbrock function or other nonlinear functions
Processing	Benchmark with other well-built code

- Learn the adjointness
- Pass dot-product test

Unit Test

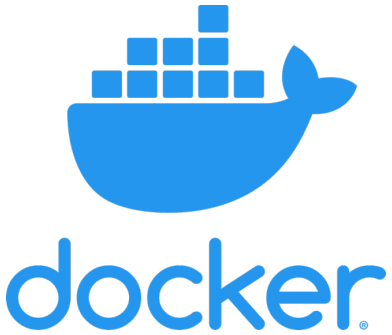
- **Unit Test in Python**
- **Write many testing cases**

```python
import unittest

# create a class for unit test
class TestStringMethods(unittest.TestCase):

    # define a method for testing
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    # define a method for testing
    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    # define a method for testing
    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        with self.assertRaises(TypeError):
            s.split(2)

# run the unit test
if __name__ == '__main__':
    unittest.main()
```

Reproducibility

But it works on my computer…

- **Docker**: OS-level virtualization to deliver software in containers
- **Singularity**: use docker images without sudo, i.e., HPC
- **Results**: Code + Parameters + Data



Miscellaneous

- **Parameter file:** json, yaml
- **Argument Parsing:** Command Line Interface
- **Logging:** DEBUG, INFO, WARNING, ERROR, and CRITICAL
- **Packaging:** pyproject.toml
- **Project Structure**

```
code
├── code
│   ├── __init__.py
│   ├── module1.py
│   └── module2.py
├── docs
│   ├── conf.py
│   ├── index.rst
│   └── module1.rst
├── tests
│   ├── __init__.py
│   ├── test_module1.py
│   └── test_module2.py
├── LICENSE
├── pyproject.toml
├── README.md
└── setup.py
```

The Role of LLM

- **Copilot**
  - VSCode extension
  - License issues
- **ChatGPT**
  - Prototype the code
  - Get ideas for coding
  - Explain other’s code

Suggestions

- **One Propagator:** One well-verified & optimized (CPU, GPU version)
- **One Solver:** One well-verified & optimized (linear & nonlinear problems)
- **Common Utilities**: data converter, processing (bp, cross-correlation), and so on.
- **Automated Scripts**:
  - Slurm job submission
  - Basic package installation (pytorch, tensorflow)
  - Data backup
  - Request cloud resources
- **Well-documented personal projects**

Recap

- Programming Environment
  - Basic Packages, Git, IDE, Anaconda, Jupyter Notebook
- Python
  - Computing in Python
  - Parallel in Python
- OOP
- Unit Test
- Reproducibility
- Miscellaneous
- LLM
- Suggestions

Thank You!