CE49X: Introduction to Computational Thinking and Data Science for Civil Engineers

Week 1: Introduction to Python and Programming Fundamentals

Dr. Eyuphan Koc

Department of Civil Engineering
Bogazici University

Fall 2025

Based on "A Whirlwind Tour of Python" by Jake VanderPlas
`https://github.com/jakevdp/WhirlwindTourOfPython`

Week 1 Outline

What is Python?

Python Origins

- Conceived in late 1980s as teaching and scripting language
- Created by Guido van Rossum
- Named after Monty Python's Flying Circus
- Now essential tool for programmers, engineers, researchers, data scientists

Why Python?

- **Simplicity and Beauty**: Clean, readable syntax
- **Versatility**: Web development, data science, automation, AI
- **Large Ecosystem**: Extensive libraries and frameworks
- **Community**: Active, supportive developer community

Python's Data Science Ecosystem: Core Libraries

Essential Data Science Libraries

- **NumPy**: Multi-dimensional arrays and mathematical operations
- **SciPy**: Scientific computing tools and algorithms
- **Pandas**: Data manipulation, analysis, and cleaning
- **Matplotlib**: 2D plotting and data visualization

Why These Matter for Civil Engineering

- Analyze structural data and sensor readings
- Process geospatial and environmental datasets
- Create engineering reports with integrated plots
- Perform statistical analysis of construction materials

Python's Data Science Ecosystem: Advanced Tools

Machine Learning and Advanced Analytics

- **Scikit-Learn**: Machine learning algorithms and tools
- **TensorFlow/PyTorch**: Deep learning frameworks
- **Jupyter**: Interactive notebooks for analysis
- **Seaborn**: Statistical data visualization

Key Insight

If there's a scientific or data analysis task you want to perform, chances are someone has written a Python package for it!

Civil Engineering Applications

- Traffic pattern analysis and optimization
- Climate impact modeling on structures

The Zen of Python

```python
import this
```

Python Philosophy (Selected)

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Readability counts
- There should be one obvious way to do it

Four Ways to Run Python Code

1. **Python Interpreter**: Interactive line-by-line execution
2. **IPython Interpreter**: Enhanced interactive environment
3. **Self-contained Scripts**: Save code in `.py` files
4. **Jupyter Notebook**: Interactive documents with code, text, and plots

Which Method to Choose?

- Quick calculations → Python/IPython interpreter
- Data analysis projects → Jupyter Notebook
- Production applications → Python scripts

Python: Interpreted vs. Compiled Languages

Python is Interpreted

Python is **interpreted**, not compiled like C/Java

- Code is executed line by line
- No separate compilation step required
- Allows interactive programming and experimentation
- Great for rapid prototyping and learning

Compiled (C/Java)

- Source $\rightarrow$ Compiler $\rightarrow$ Executable
- Faster execution
- Catch errors early

Interpreted (Python)

- Source $\rightarrow$ Interpreter
- Interactive development
- Runtime flexibility

1. Python Interpreter

Starting Python

```
$ python
Python 3.9.7 (default, Sep 16 2021, 16:13:09)
>>>
```

```
>>> 1 + 1
2
>>> x = 5
>>> x * 3
15
```

- Great for quick calculations and testing
- Uses »> prompt
- Limited editing capabilities

2. IPython Interpreter

Enhanced Python Experience

```
$ ipython
IPython 8.0.0 -- An enhanced Interactive Python.
In [1]:
```

```
In [1]: 1 + 1
Out[1]: 2

In [2]: x = 5

In [3]: x * 3
Out[3]: 15
```

- Numbered input/output
- Tab completion, syntax highlighting
- Magic commands (%time, %run, etc.)

3. Self-contained Scripts

Creating a Python Script

Create file `test.py`:

```python
# file: test.py
print("Running test.py")
x = 5
print("Result is", 3 * x)
```

Running the Script

```
$ python test.py
Running test.py
Result is 15
```

- Best for longer programs
- Can be reused and shared

4. Jupyter Notebook

Interactive Computing Environment

- Combines code, text, equations, and visualizations
- Web-based interface
- Supports multiple programming languages
- Excellent for data analysis and research

Features

- Rich text with Markdown
- Inline plots and graphics
- Easy sharing and collaboration
- Export to various formats (PDF, HTML, etc.)

Syntax vs. Semantics

Definition (Syntax vs. Semantics).

- ***Syntax***: *Structure of the language (what constitutes correct code)*
- ***Semantics***: *Meaning of the code (what the code actually does)*

Python as "Executable Pseudocode"

Python's clean syntax makes it often easier to read than other languages like C or Java

Example: Basic Python Script

```python
# set the midpoint
midpoint = 5

# make two empty lists
lower = []; upper = []

# split the numbers into lower and upper
for i in range(10):
    if (i < midpoint):
        lower.append(i)
    else:
        upper.append(i)

print("lower:", lower)
print("upper:", upper)

#output = lower: [0, 1, 2, 3, 4]
#output = upper: [5, 6, 7, 8, 9]
```

1. Comments Are Marked by

```python
# This is a standalone comment
x = 5  # This is an inline comment
```

- Everything after # is ignored by interpreter
- Can be standalone or inline
- No multi-line comment syntax (use triple quotes for docstrings)

Good Practice

```python
x += 2  # shorthand for x = x + 2
```

2. End-of-Line Terminates Statement

```python
midpoint = 5  # No semicolon needed!
```

Line Continuation

Use backslash \ or parentheses for long statements:

```python
# Method 1: Backslash (discouraged)
x = 1 + 2 + 3 + 4 +\
    5 + 6 + 7 + 8

# Method 2: Parentheses (preferred)
x = (1 + 2 + 3 + 4 +
     5 + 6 + 7 + 8)
```

3. Semicolon Can Optionally Terminate

```
# Multiple statements on one line
lower = []; upper = []

# Equivalent to:
lower = []
upper = []
```

Style Recommendation

Generally discouraged by Python style guides (PEP 8)
Use separate lines for better readability

4. Indentation: Whitespace Matters!

C Language (Braces)

```c
for(int i=0; i<100; i++)
{
    // curly braces indicate block
    total += i;
}
```

Python (Indentation)

```python
for i in range(100):
    # indentation indicates block
    total += i
```

Critical Rule

Indented code blocks are always preceded by a colon (:)

Indentation Examples

Code Block Inside

```
if x < 4:
    y = x * 2
    print(x)  # Inside block
```

Code Block Outside

```
if x < 4:
    y = x * 2
print(x)  # Outside block
```

Indentation Rules

- Amount of indentation is flexible (but be consistent!)
- Convention: 4 spaces per indentation level
- Most editors can auto-indent Python code

5. Whitespace Within Lines

All Equivalent

```
x=1+2
x = 1 + 2
x                    =            1      +                        2
```

Readability Matters

Compare:

```
x=10**-2            # Hard to read
x = 10 ** -2        # Much clearer!
```

PEP 8 Style Guide

Use single space around binary operators, no space around unary operators

6. Parentheses: Grouping and Calling

Mathematical Grouping

```
result = 2 * (3 + 4)  # = 14, not 10
```

Function Calls

```
print('Hello, World!')
print('first value:', 1)
```

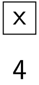
Method Calls

```
L = [4, 2, 3, 1]
L.sort()  # Parentheses required even with no arguments
print(L)  # [1, 2, 3, 4]
```

Python Variables Are Pointers

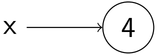
C Language (Containers)

```c
int x = 4;   // x is a memory bucket
```



Python (Pointers)

```python
x = 4   # x points to object 4
```



Dynamic Typing

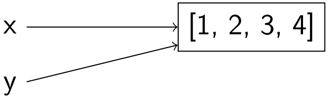
Variables can point to objects of any type:

```python
x = 1          # x is an integer
x = 'hello'    # now x is a string
x = [1, 2, 3]  # now x is a list
```

Pointer Behavior: Mutable Objects

```python
x = [1, 2, 3]
y = x            # Both point to same list
print(y)         # [1, 2, 3]

x.append(4)      # Modify through x
print(y)         # [1, 2, 3, 4] - y changed too!
```



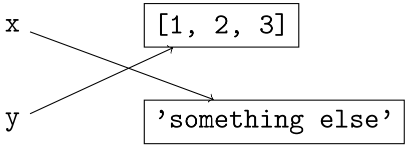
Key Insight

When two variables point to the same mutable object, changes through one affect the other!

Reassignment vs. Modification

```python
x = [1, 2, 3]
y = x

x = 'something else'   # Reassignment
print(y)               # [1, 2, 3] - y unchanged
```



Immutable Objects

Numbers, strings, and tuples are immutable - safe from this behavior:

```python
x = 10; y = x; x += 5
print(f"x = {x}, y = {y}")  # x = 15, y = 10
```

Everything Is an Object

```
x = 4
type(x)          # <class 'int'>

x = 'hello'
type(x)          # <class 'str'>

x = 3.14159
type(x)          # <class 'float'>
```

Objects Have Attributes and Methods

```
L = [1, 2, 3]
L.append(100)   # Method call
print(L)         # [1, 2, 3, 100]

x = 4.5
print(x.real, "+", x.imag, "i")  # 4.5 + 0.0 i
```

Even Simple Types Have Methods

```python
x = 4.5
x.is_integer()  # False

x = 4.0
x.is_integer()  # True

# Even methods are objects!
type(x.is_integer)  # <class 'method'>
```

Everything-is-Object Philosophy

This design choice enables very convenient language constructs and powerful introspection capabilities

Basic Arithmetic Operators

Operator	Name	Description
`a + b`	Addition	Sum of a and b
`a - b`	Subtraction	Difference of a and b
`a * b`	Multiplication	Product of a and b
`a ** b`	Exponentiation	a raised to power b
`-a`	Negation	Negative of a
`+a`	Unary plus	a unchanged

```python
# Basic arithmetic examples
print(10 + 5)   # 15 (addition)
print(10 - 3)   # 7 (subtraction)
print(4 * 6)    # 24 (multiplication)
print(2 ** 3)   # 8 (exponentiation)
```

Division and Modulus Operators

Operator	Name	Description
`a / b`	True division	Quotient of a and b (float result)
`a // b`	Floor division	Integer quotient (rounded down)
`a % b`	Modulus	Remainder after division

```python
# Division examples
print(25 / 4)    # 6.25 (true division - always float)
print(25 // 4)   # 6 (floor division - integer result)
print(25 % 4)    # 1 (modulus - remainder)

# Useful for checking even/odd numbers
print(7 % 2)     # 1 (odd number)
print(8 % 2)     # 0 (even number)
```

Comparison Operators

Operator	Description	Example
`a == b`	Equal to	`5 == 5` → `True`
`a != b`	Not equal to	`5 != 3` → `True`
`a < b`	Less than	`3 < 5` → `True`
`a <= b`	Less than or equal	`3 <= 3` → `True`
`a > b`	Greater than	`5 > 3` → `True`
`a >= b`	Greater than or equal	`5 >= 5` → `True`

```python
# Chained comparisons (unique to Python!)
x = 5
print(1 < x < 10)     # True
print(10 < x < 20)    # False
```

Boolean Operators

Operator	Description	Example
`a and b`	Logical AND	`True and False` $\rightarrow$ `False`
`a or b`	Logical OR	`True or False` $\rightarrow$ `True`
`not a`	Logical NOT	`not True` $\rightarrow$ `False`

```python
# Short-circuit evaluation
x = 5
print(x > 0 and x < 10)  # True
print(x > 10 or x < 0)   # False

# Identity and membership
print(x is None)         # False
print(x in [1, 2, 5])    # True
```

Assignment Operators

Standard Assignment

```python
x = 5
x = x + 3  # x becomes 8
```

Augmented Assignment

```python
x = 5
x += 3      # x becomes 8
```

Operator	Equivalent
`x += a`	`x = x + a`
`x -= a`	`x = x - a`
`x *= a`	`x = x * a`
`x /= a`	`x = x / a`
`x //= a`	`x = x // a`
`x %= a`	`x = x % a`
`x **= a`	`x = x ** a`

Python's Built-in Scalar Types

Type	Example	Description
`int`	`x = 1`	Integers (whole numbers)
`float`	`x = 1.0`	Floating-point numbers
`complex`	`x = 1 + 2j`	Complex numbers
`bool`	`x = True`	Boolean values
`str`	`x = 'abc'`	Text strings
`NoneType`	`x = None`	Null value

Dynamic Typing

Python automatically determines the type based on the value assigned

Type Checking and Conversion

```python
# Type checking
print(type(42))         # <class 'int'>
print(type(3.14))       # <class 'float'>
print(type(True))       # <class 'bool'>
print(type("hello")) # <class 'str'>

# Type conversion
print(int(3.14))        # 3 (convert float to int)
print(float(42))        # 42.0 (convert int to float)
print(str(123))         # '123' (convert int to string)
print(bool(1))          # True (convert int to bool)
```

Integers

```python
# Integer literals
a = 42            # Decimal
b = 0b101010      # Binary (42 in decimal)
c = 0o52          # Octal (42 in decimal)
d = 0x2a          # Hexadecimal (42 in decimal)

print(a, b, c, d)  # All print 42

# Python 3 integers have unlimited precision!
big_num = 2 ** 1000
print(len(str(big_num)))  # 302 digits!
```

Python 2 vs 3

Python 2 had separate `int` and `long` types
Python 3 unified them into a single `int` type

Floating-Point Numbers

```python
# Float literals
x = 1.0
y = 1.
z = .5
w = 1e10     # Scientific notation: 1 * 10^10
v = 1.5e-3   # 1.5 * 10^-3 = 0.0015

print(x, y, z, w, v)
```

Floating-Point Precision

```python
print(0.1 + 0.2)  # 0.30000000000000004
print(0.1 + 0.2 == 0.3)  # False!
```

Use `math.isclose()` for floating-point comparisons

Complex Numbers

```python
# Complex number literals
z1 = 1 + 2j
z2 = complex(3, 4)  # 3 + 4j

print(z1.real)  # 1.0
print(z1.imag)  # 2.0
print(abs(z1))  # 2.23606797749979 (magnitude)

# Complex arithmetic
z3 = z1 + z2
print(z3)       # (4+6j)
```

Note

Use `j` or `J` for imaginary unit (not `i` like in mathematics)

Boolean Values

```python
# Boolean literals
flag1 = True
flag2 = False

# Boolean operations
print(True and False)    # False
print(True or False)     # True
print(not True)          # False

# Booleans are subclass of int!
print(True + False)      # 1
print(True * 5)          # 5
```

Boolean Values

Truthiness in Python

Many objects can be evaluated as True/False:

```python
print(bool(0))        # False
print(bool(42))       # True
print(bool(""))       # False (empty string)
print(bool("hi"))     # True (non-empty string)
```

Strings: Literals and Operations

```python
# String literals
s1 = 'single quotes'
s2 = "double quotes"
s3 = '''triple quotes
allow multiple lines'''

# String operations
name = "Python"
print(len(name))          # 6
print(name[0])            # 'P' (indexing)
print(name.upper())       # 'PYTHON'
print(name.lower())       # 'python'
```

String Formatting

```python
# Modern string formatting
age = 25
name = "Alice"

# f-strings (Python 3.6+) - Recommended
print(f"I am {name}, {age} years old")

# .format() method (older Python versions)
print("I am {}, {} years old".format(name, age))

# % formatting (legacy)
print("I am %s, %d years old" % (name, age))
```

Best Practice

Use f-strings for Python 3.6+ - they're faster and more readable!

None Type

```python
# None represents absence of value
x = None
print(x)              # None
print(type(x))        # <class 'NoneType'>

# Common usage
def greet(name=None):
    if name is None:
        name = "World"
    print(f"Hello, {name}!")

greet()           # Hello, World!
greet("Alice")  # Hello, Alice!
```

None Type

Important

Use `is` and `is not` when comparing with `None`:

```python
if x is None:        # Correct
if x is not None:    # Correct
if x == None:        # Works but not recommended
```

Python's Compound Types

Type	Example	Description
`list`	`[1, 2, 3]`	Ordered, mutable collection
`tuple`	`(1, 2, 3)`	Ordered, immutable collection
`dict`	`{'a':1, 'b':2}`	Key-value mapping
`set`	`{1, 2, 3}`	Unordered unique values

Bracket Types Matter!

- Square brackets `[]` → lists
- Round brackets `()` → tuples
- Curly brackets `{}` → dictionaries or sets

Lists: Creating and Accessing

```python
# Creating lists
primes = [2, 3, 5, 7, 11]
mixed = [1, 'hello', 3.14, True]
empty = []

# List operations
print(len(primes))      # 5
print(primes[0])        # 2 (first element)
print(primes[-1])       # 11 (last element)
print(primes[1:3])      # [3, 5] (slicing)
```

Lists: Modifying Content

```python
primes = [2, 3, 5, 7, 11]

# Modifying lists
primes.append(13)         # Add to end
primes.insert(0, 1)       # Insert at position
primes.remove(1)          # Remove first occurrence
print(primes)             # [2, 3, 5, 7, 11, 13]
```

Key Points

- Lists are **mutable** - can be changed after creation
- Support indexing (`list[0]`) and slicing (`list[1:3]`)
- Negative indices count from end (`list[-1]`)

List Methods

```python
numbers = [3, 1, 4, 1, 5, 9, 2, 6]

# Common methods
numbers.sort()              # Sort in place
print(numbers)              # [1, 1, 2, 3, 4, 5, 6, 9]

numbers.reverse()           # Reverse in place
print(numbers)              # [9, 6, 5, 4, 3, 2, 1, 1]

print(numbers.count(1)) # 2 (count occurrences)
print(numbers.index(5)) # 2 (find first index)
```

List Concatenation

```python
# Combining lists
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Method 1: Create new list
combined = list1 + list2     # [1, 2, 3, 4, 5, 6]

# Method 2: Modify existing list
list1.extend(list2)          # list1 becomes [1, 2, 3, 4, 5, 6]
print(list1)
```

Important Difference

+ creates a new list, `extend()` modifies the original list

Tuples: Ordered and Immutable

```python
# Creating tuples
point = (3, 4)
colors = ('red', 'green', 'blue')
single = (42,)              # Note the comma!
empty = ()

# Tuple operations (read-only)
print(len(point))           # 2
print(point[0])             # 3
print(point[1])             # 4

# Tuple unpacking
x, y = point
print(f"x={x}, y={y}")  # x=3, y=4

# Multiple assignment
a, b, c = colors
print(a, b, c)              # red green blue
```

Dictionaries: Key-Value Mapping

```python
# Creating dictionaries
student = {'name': 'Alice', 'age': 20, 'major': 'CS'}
grades = {'math': 95, 'physics': 87, 'chemistry': 92}
empty = {}

# Dictionary operations
print(student['name'])          # 'Alice'
print(len(student))             # 3
print('age' in student)         # True

# Modifying dictionaries
student['gpa'] = 3.8            # Add new key-value
student['age'] = 21             # Update existing
del student['major']            # Remove key-value

print(student.keys())           # dict_keys(['name', 'age', 'gpa'])
print(student.values())         # dict_values(['Alice', 21, 3.8])
```

Sets: Creating and Basic Operations

```python
# Creating sets
vowels = {'a', 'e', 'i', 'o', 'u'}
numbers = {1, 2, 3, 3, 4, 4, 5}  # Duplicates removed
print(numbers)                   # {1, 2, 3, 4, 5}

# Set methods
vowels.add('y')                  # Add element
vowels.remove('a')               # Remove (KeyError if not found)
vowels.discard('z')              # Remove (no error if not found)
print(vowels)
```

Key Feature

Sets automatically remove duplicates and are unordered

Set Operations

```python
# Mathematical set operations
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

print(set1 | set2)          # {1, 2, 3, 4, 5, 6} (union)
print(set1 & set2)          # {3, 4} (intersection)
print(set1 - set2)          # {1, 2} (difference)
print(set1 ^ set2)          # {1, 2, 5, 6} (symmetric difference)
```

Set Operation Symbols

- | = Union (all elements)
- & = Intersection (common elements)
- - = Difference (in first, not second)
- ^ = Symmetric difference (not in both)

What We've Covered

Python Basics

- Python philosophy and ecosystem
- Four ways to run Python code
- Syntax fundamentals
- Comments, indentation, whitespace

Variables & Objects

- Variables as pointers
- Dynamic typing
- Mutable vs immutable objects
- Everything-is-object philosophy

Operators & Types

- Arithmetic, comparison, boolean operators
- Scalar types: int, float, complex, bool, str, None
- Type checking and conversion

Data Structures

- Lists: ordered, mutable
- Tuples: ordered, immutable
- Dictionaries: key-value mapping
- Sets: unique values

Key Takeaways

Python's Core Principles

- **Readability counts** - Code should be easy to understand
- **Simple is better than complex** - Prefer clear solutions
- **Everything is an object** - Consistent behavior across types

Next Week Preview

- Control flow: if/elif/else, for/while loops
- Functions: definition, arguments, return values
- Error handling: try/except/finally
- Iterators and list comprehensions

Practice Exercises

1. Create a list of your favorite programming languages and sort them alphabetically
2. Write a dictionary mapping country names to their capitals
3. Use set operations to find common elements between two lists
4. Practice string formatting with f-strings and the .format() method
5. Experiment with different number bases (binary, octal, hexadecimal)

Resources

- Original notebook: `https://github.com/jakevdp/WhirlwindTourOfPython`
- Python documentation: `https://docs.python.org/3/`
- PEP 8 Style Guide: `https://www.python.org/dev/peps/pep-0008/`

Questions?

Thank you!

Dr. Eyuphan Koc
eyuphan.koc@bogazici.edu.tr

Next Week: Control Flow, Functions, and Error Handling