Python Scripting: Introduction & Fundamentals

1. Introduction

1.1 Audience Spectrum

This lecture is designed for a mixed audience: some already experienced with Python, others complete beginners. The approach balances basic teaching with deeper insights, aiming to appeal to both groups.

1.2 Teaching Approach

- Uses live examples to build intuition.
- Encourages Pythonic practices and high-level thinking.
- Highlights common pitfalls, historical motivations, and the philosophy behind Python features.

---

2. Writing and Executing Python

2.1 Interactive Use

Python can be run in various environments:

- Emacs (`Meta-X run-python`)
- Terminal/command line
- Jupyter notebooks
- Any preferred editor or IDE

Table: Python Execution Environments

Environment	How to Start	Features
REPL (python)	`python` in terminal	Interactive, quick testing
Script	`python script.py`	Runs full file, reproducible
Jupyter Notebook	`jupyter notebook`	Rich output, cells, visualization
IDE (e.g., PyCharm)	Open project, run	Debugging, code completion
Emacs	`Meta-X run-python`	Editor integration

2.2 Running Scripts vs. Interactive Mode

- **Script mode:** Good for reproducibility, sharing, and larger programs.
- **Interactive mode:** Good for experimentation, debugging, and learning.

2.3 Example: Parsing Stock Market Data

Given a string:

```
"GOOG,100,153.36"
```

Goal:

- Extract the symbol ("GOOG")
- Convert 100 to int
- Convert 153.36 to float

Target Output:

```
['GOOG', 100, 153.36]
```

Step-by-step:

1. **Split the string** using `.split(',')`:

   ```
   s_line = "GOOG,100,153.36".split(',')  # ['GOOG', '100', '153.36']
   ```

2. **Define types**:

```
types = [str, int, float]
```

3. **Zip types and values**:

```
list(zip(types, s_line))  # [(str, 'GOOG'), (int, '100'), (float, '153.36')]
```

   - Note: `zip` returns an iterator; casting to `list()` is needed to see all elements.
   - Once iterated, a zip object is exhausted.

4. **List comprehension for conversion**:

```
[c(v) for c, v in zip(types, s_line)]  # ['GOOG', 100, 153.36]
```

Pythonic Principle

- Prefer list comprehensions and high-level constructs over manual indexing and C++-style loops.

---

3. Pythonic Programming Philosophy

3.1 Avoid Low-Level Thinking

- Avoid C++-style manual indexing, pointer manipulation, and low-level constructs.
- Python encourages thinking at a higher level of abstraction.

3.2 Embrace High-Level Abstractions

- Use list comprehensions, built-in functions, and idiomatic constructs.
- Prefer `for x in y` over `for i in range(len(y))`.

---

4. Brief History of Python

4.1 Language Evolution

- **Fortran (1950s):** Unforgiving, used for scientific computing, not beginner-friendly.
- **BASIC (1960s):** Simplified Fortran, beginner-friendly, used timesharing.
- **ABC (1980s, Netherlands):**
  - Enforced indentation for block structure.
  - Built-in data structures (no need to implement basic algorithms like heapsort).
  - Focused on scripting and ease of use.
  - Flopped commercially, but ideas lived on.
- **Python:**
  - Designed as a scripting language with built-in data structures and enforced indentation.
  - Aimed to be simple, readable, and educational.
  - Influenced by frustration with poor-quality scripting tools (e.g., Perl).

---

5. Python Syntax and Indentation

5.1 Indentation Rules

- Blocks start with a colon (`:`):

```
if x > 0:
    print("Positive")
```

- All code in a block must be indented equally.
- Mixing tabs and spaces is discouraged; use spaces only.
- Copy-paste errors can cause indentation issues.

- Indentation is enforced by the interpreter; improper indentation leads to syntax errors.

Tabs vs Spaces

- Tabs can be interpreted differently (4 vs 8 spaces); avoid them.
- Use single ASCII spaces for indentation.

Metaprogramming Caveat

- Indentation-based syntax makes it harder to write programs that generate other programs (metaprogramming).

---

6. Numerical Types and Operations

6.1 Integers

- Arbitrary precision (limited by available memory).
- Example:

```
x = 10**1000  # Very large integer
```

6.2 Floats

- Limited to ~10^308; overflow produces `inf`.

```
float('inf') > 99999999  # True
```

6.3 Division

- `/` is floating-point division:

```
1 / 2  # 0.5
```

- `//` is integer (floor) division:

```
1 // 2  # 0
```

6.4 Complex Numbers

- Use `j` for the imaginary part:

```
1 + 2j
import cmath
cmath.sqrt(-1)  # returns 1j
```

- `cmath` for complex math, `math` for real math.

---

7. Strings in Python

7.1 Quotes

- Single `'abc'` or double `"abc"` quotes are interchangeable.
- Python displays with single quotes by default.

7.2 Escape Sequences

- `\n` for newline, `\t` for tab, etc.

7.3 Raw Strings

- Prefix with `r` to prevent escape sequence interpretation:

```
r"\n"  # literal backslash and n
```

7.4 Triple Quotes

- Use triple quotes for multi-line strings:

```
'''This is
```

a multiline string'''

````
- Triple double quotes (`"""..."""`) also allowed, but don't mix single and double.

---

## 8. Python Object Model

### 8.1 Every Object Has:
- **Identity:** via `id(obj)` (like a pointer/address, but not directly usable)
- **Type:** via `type(obj)`
- **Value:** the object itself

### 8.2 Immutability
- **Immutable:** Value cannot change (`int`, `str`, `float`, etc.)
- **Mutable:** Value can change (`list`, `dict`, etc.)

#### Example:
```python
a = 12
print(id(a))
a += 1
print(id(a))  # ID has changed
````

8.3 Variables vs Objects

- Variables are bindings to objects, not the objects themselves.
- Assignment copies references, not objects.

Example:

```
a = [1, 2, 3]
b = a
b.append(4)
print(a)  # [1, 2, 3, 4]
```

8.4 Aliasing and Identity

- Assigning one variable to another copies the reference.
- `is` checks identity (like pointer comparison in C++), `==` checks value equality.

Example Table:

Operation	Description
`a is b`	True if `a` and `b` are the same object
`a == b`	True if `a` and `b` have equal values

---

9. Python's Built-In Data Types Overview

9.1 `None`

- Singleton object representing "nothing" (like `null` or `nullptr`).

9.2 Numbers
```

- `int`, `float`, `complex`

9.3 Sequences

- Common: `str`, `list`, `tuple`
- Indexable, iterable

9.4 Mappings

- Primarily `dict`: key/value store

9.5 Callables

- Functions, methods, classes, lambdas

---

10. Python Sequences

10.1 Indexing

- `s[i]` returns the ith element (0-based)
- Raises `IndexError` if out of range (unlike C++, which is undefined behavior)

10.2 Negative Indexing

- `s[-1]` is the last element, `s[-2]` is the second-to-last, etc.
- Valid for `-len(s) <= i < len(s)`

10.3 Slicing

- `s[i:j]` returns a new sequence from index i up to (but not including) j
- `s[:j]` from start to j-1
- `s[i:]` from i to end
- `s[:]` is a copy of the entire sequence
- Slices are new objects, but elements are references to the same objects
- If `i == j`, returns empty sequence; if `i > j`, raises error

ASCII Diagram: Slicing

```
Sequence:  S = [A, B, C, D, E]
Indexes:    0  1  2  3  4
Negative:  -5 -4 -3 -2 -1

S[1:4] -> [B, C, D]
S[-3:-1] -> [C, D]
```

10.4 Common Sequence Operations

Operation	Description
`len(s)`	Number of elements
`min(s)`	Minimum element
`max(s)`	Maximum element
`list(s)`	Convert sequence to list

- `min`/`max` require comparable elements; error if types are mixed or sequence is empty.
- Sequences can be heterogeneous, but many operations expect homogeneity.

10.5 Strings Are Immutable

- Cannot assign to elements or slices of a string.
- To modify, convert to list, change, then join back to string.

---

11. Mutable Sequences: Lists

11.1 Assign to Element

```
lst[i] = value
```

11.2 Assign to Slice

```
lst[i:j] = [a, b]
```

- Can grow or shrink the list.
- Assignment replaces the slice with the new sequence.

11.3 Delete Items

```
del lst[i]        # Delete index
del lst[i:j]      # Delete slice
```

- Decreases the length of the list.
- `del` only works on mutable sequences.

11.4 List-Specific Methods

Method	Description
`append(v)`	Add element to end
`extend([v])`	Add multiple elements to end
`insert(i, v)`	Insert before index i
`pop()`	Remove and return last item
`pop(i)`	Remove and return item at index i
`count(v)`	Count occurrences
`index(v)`	Index of first matching item
`sort()`	In-place sort

- `append` is amortized O(1) due to over-allocation.
- `extend` is a batch append.
- `insert` and `pop` can operate at any index.
- `sort` is in-place and uses a stable algorithm (Timsort).

ASCII Diagram: List Append and Growth

```
Initial:   [A, B, C]
append(D): [A, B, C, D]

Internal (over-allocated):
[ A | B | C | D |   |   ]
                ^
             (unused slots)
```

Amortized Analysis of Append

- Python lists over-allocate memory (double when full).
- Most appends are O(1); occasional O(n) when resizing.
- Total cost of n appends: O(n).

---

12. Iteration and Comprehensions

12.1 Iterating Over Sequences

- Use `for x in sequence` to iterate over elements.
- Avoid manual indexing unless necessary.

12.2 List Comprehensions

- Concise way to build lists:

```
[f(x) for x in sequence]
```

- More Pythonic than manual loops.

---

13. Summary

This lecture introduced Python from both basic and intermediate perspectives. It covered:

- Python's core types and data structures
- The distinction between mutable and immutable objects
- Data parsing using list comprehensions and functions like `zip()`
- The unique object model of Python (identity, type, value)
- Sequences (strings, lists, tuples): indexing, slicing, mutation
- Efficiency of list operations (especially `append()` and amortized analysis)
- Pythonic idioms and reasoning for clean, efficient code

---

14. Additional Notes and Edge Cases

- **Strings and Lists:**
  - To convert a list of characters back to a string: `''.join(list_of_chars)`
- **Equality:**
  - `==` compares values, `is` compares identities.
- **Slicing with Steps:**
  - `s[i:j:k]` allows stepping; negative steps reverse direction.
- **Error Handling:**
  - Indexing out of range raises `IndexError`.
  - `min`/`max` on empty or non-comparable sequences raises errors.
- **List Growth:**
  - Assigning to `lst[len(lst):]` is equivalent to `append`.
- **Mutable vs Immutable:**
  - Mutating a mutable object does not change its identity.
  - You cannot change the type of an object after creation.

---

15. Visual Summary

Python Object Model (ASCII Diagram)

```
Variable   --->   Object (has id, type, value)
   |
   +--->   [id(obj)]
   +--->   [type(obj)]
   +--->   [value]

Assignment:
  a = obj1
  b = a   # b points to the same object as a

Aliasing:
  a = [1,2,3]
  b = a
  b.append(4)
  # a and b both see [1,2,3,4]
```

List Growth (ASCII Diagram)

```
[ A | B | C ]  --append(D)-->  [ A | B | C | D |   |   ]
```

2.3 Error Handling and Exceptions

Python uses exceptions to handle errors. If an error occurs, Python raises an exception and (unless caught) stops execution.

Example: Handling Exceptions

```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except ValueError:
    print("Not a valid integer!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Table: Common Python Exceptions

Exception	When It Occurs
`ValueError`	Invalid value (e.g., `int('abc')`)
`TypeError`	Wrong type (e.g., `1 + 'a'`)
`IndexError`	Sequence index out of range
`KeyError`	Dict key not found
`ZeroDivisionError`	Division by zero
`AttributeError`	Attribute not found
`ImportError`	Import fails
`FileNotFoundError`	File does not exist

- Use `raise` to throw exceptions manually.
- Use `finally` for cleanup code that always runs.

2.4 Functions: Definition, Arguments, and Scope

Defining Functions

```
def add(x, y=0):
    return x + y
```

- `x` is a required argument, `y` has a default value.

Argument Types

Type	Example	Description
Positional	`f(1, 2)`	Matched by position
Keyword	`f(x=1, y=2)`	Matched by name
Default	`def f(x, y=0)`	Uses default if not provided
*args	`def f(*args)`	Tuple of extra positional args
**kwargs	`def f(**kwargs)`	Dict of extra keyword args

Example: All Argument Types

```
def demo(a, b=2, *args, **kwargs):
    print(a, b, args, kwargs)
```

Variable Scope Diagram

```
```

```
[Global Scope]
    |
    +--[Enclosing Scope]
           |
           +--[Local Scope]
```

- Python uses LEGB rule: Local, Enclosing, Global, Built-in.

2.5 Mutable vs. Immutable Types

Type	Mutable?	Examples
int, float	No	1, 2.5
str, tuple	No	'abc', (1, 2)
list, dict	Yes	[1, 2], {'a': 1}
set	Yes	{1, 2, 3}

Pitfall: Default mutable arguments retain changes between calls.

```
def f(x, lst=[]):
    lst.append(x)
    return lst
f(1)  # [1]
f(2)  # [1, 2]  # Not what you expect!
```

- Use `None` as default and create inside function.

2.6 Advanced List Comprehensions

- With condition:

  ```
  [x for x in range(10) if x % 2 == 0]  # [0, 2, 4, 6, 8]
  ```

- Nested:

  ```
  [(x, y) for x in range(2) for y in range(2)]  # [(0,0), (0,1), (1,0), (1,1)]
  ```

Construct	Example	Output
List comprehension	`[x*x for x in range(3)]`	`[0, 1, 4]`
Map	`list(map(lambda x: x*x, range(3)))`	`[0, 1, 4]`
Filter	`list(filter(lambda x: x%2, range(5)))`	`[1, 3]`

- List comprehensions are generally more readable and Pythonic.

2.7 Dictionaries and Sets

Dictionaries (`dict`)

- Key-value store, fast lookup.

```
d = {'a': 1, 'b': 2}
d['c'] = 3
print(d['a'])  # 1
```

Method	Description

Method	Description
`d.keys()`	All keys
`d.values()`	All values
`d.items()`	All (key, value) pairs
`d.get(k, v)`	Get value or default
`d.pop(k)`	Remove and return value

Sets (`set`)

- Unordered, unique elements.

```
s = {1, 2, 3}
s.add(4)
s.remove(2)
```

Method	Description
`add(x)`	Add element
`remove(x)`	Remove element
`union(s2)`	Union with another set
`intersection(s2)`	Intersection with another set

2.8 String Formatting

- Old style: `'Hello %s' % name`
- `str.format()`: `'Hello {}'.format(name)`
- f-strings (Python 3.6+): `f'Hello {name}'`

Example Table

Method	Example	Output
`%`	`'%d %s' % (3, 'hi')`	`'3 hi'`
`.format()`	`'{} {}'.format(3, 'hi')`	`'3 hi'`
f-string	`f'{3} hi'`	`'3 hi'`

2.9 Modules and Imports

- Use `import module` to use standard library or your own code.
- Example:

  ```
  import math
  print(math.sqrt(16))  # 4.0
  from math import pi
  print(pi)
  ```

- Use `as` to alias: `import numpy as np`

2.10 Iterators and Generators

- Any object with `__iter__()` and `__next__()` is an iterator.
- Use `iter()` to get an iterator, `next()` to get next value.

Example: Manual Iteration

```
lst = [1, 2, 3]
it = iter(lst)
```

```
print(next(it))  # 1
print(next(it))  # 2
```

Generator Functions

- Use `yield` to create a generator.

```
def count_up(n):
    i = 0
    while i < n:
        yield i
        i += 1
for x in count_up(3):
    print(x)  # 0 1 2
```

2.11 Pythonic Best Practices and Anti-Patterns

Practice	Pythonic?	Example/Note
List comprehensions	Yes	`[x*x for x in xs]`
Manual indexing	No	`for i in range(len(xs))`
Use of built-in functions	Yes	`sum(xs)`, `min(xs)`
C-style loops	No	`for (int i=0; i<n; i++)`
Exception handling	Yes	`try/except`
Catch-all except (bare except)	No	`except:` (should specify exception)
Mutable default arguments	No	`def f(x, lst=[])` (use `None` instead)
Use of `is` for equality	No	Use `==` for value equality
Use of `is` for identity	Yes	Use `is` for singleton checks (`None`)
Explicit variable naming	Yes	`total_sum` vs. `ts`
Wildcard imports (`from x import *`)	No	Pollutes namespace, unclear origin