Python Introduction & Fundamentals

1. Introduction

1.1 Audience Spectrum

The lecture is aimed at a mixed audience: some already experienced with Python and some beginners. The approach balances basic teaching with deeper insights to appeal to both groups.

1.2 Teaching Approach

- Uses live examples to build intuition.
- Encourages embracing Pythonic practices.
- Highlights common pitfalls and historical motivations behind Python features.

---

2. Writing and Executing Python

2.1 Interactive Use

Python can be run in various environments; the example uses Emacs (`Meta-X run-python`), but any environment such as Jupyter or terminal is acceptable.

2.2 Example Scenario: Parsing Stock Market Data

Given a string like:

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

Goal: - Extract the symbol (“GOOG”) - Convert 100 to int - Convert 153.36 to float

Target Output:

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

---

3. Python Types and Data Parsing

3.1 Python Basic Data Types

- `str`: a string
- `int`: an integer
- `float`: a floating point number

3.2 Parsing with Types

Python has constructor functions (which are actually classes) to cast strings:

```
[int("100"), float("153.36")]  # Converts strings to specified types
```

3.3 The `zip()` Function

Zips two sequences element-wise:

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

- Lazily evaluated: must be cast to `list()` to see all elements.
- Reusing iterated zip objects yields empty results.

3.4 List Comprehensions

Powerful Python feature for transforming data:

Example:

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

Benefits: - Concise expression of logic - More “Pythonic” than index-based loops

---

4. Pythonic Programming Philosophy

4.1 Avoid Low-Level Thinking

Avoid thinking like in C++ (manual indexing, mutable pointers, etc.)

4.2 Think High-Level

Utilize Python abstractions such as: - List comprehensions - Built-in functions - Prefer `for x in y` over `for i in range(len(y))`

---

5. Brief History of Python

5.1 Fortran → BASIC → ABC → Python

- Fortran (1950s): Popular but unforgiving
- BASIC (1960s): Beginner friendly
- ABC (1980s): Dutch origins, focused on ease, enforced indentation

5.2 ABC Concepts Carried To Python:

- Indentation used for block structure (not braces)
- Built-in data structures
- Focus on scripting, ad-hoc tasks
- Prefixes simplicity, readability, and education
- Also influenced by frustration with poor-quality scripting tools like Perl

---

6. Python Syntax and Indentation

6.1 Important Syntax Rule:

Blocks start with a colon:

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

6.2 Copy-Paste Issues

- Copying improperly indented code can cause syntax errors.
- Maintain consistent indent spacing.

6.3 Tabs vs Spaces

- Stick to spaces for indentation.
- Avoid mixing tabs and spaces.
- Recommendation: Never use tabs to avoid compatibility issues.

---

7. Numerical Types and Operations

7.1 Integers

- Arbitrary precision
- You can do `10**1000` without overflow.

7.2 Floats

- Produces `inf` when overflowed

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

7.3 Division

```
1 / 2   # 0.5 (float division)
1 // 2  # 0 (integer division)
```

7.4 Complex Numbers

- Use `j` for imaginary part:

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

---

8. Strings in Python

8.1 Quotes

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

8.2 Escape Sequences

```
"\n"  # newline
"\t"  # tab
```

8.3 Raw Strings

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

8.4 Triple Quotes

Use triple quotes for multi-line strings:

```
'''This is
a multiline string'''
```

---

9. Python Object Model

9.1 Every Object Has:

- Identity → via `id(obj)`
- Type → via `type(obj)`
- Value → the object itself

9.2 Immutability

- Immutable: Value cannot change (`int`, `str`, `float`)
- Mutable: Value can change (`list`, `dict`, ...)

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

9.3 Variables ≠ Objects

Variables are bindings to objects, not the objects themselves.

9.4 Aliasing

```
a = [1, 2, 3]
```

```
print(a)  # [1, 2, 3, 4]
```

Both refer to the same list.

---

10. Python's Built-In Data Types Overview

10.1 `None`

Singleton object representing "nothing".

10.2 Numbers

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

10.3 Sequences

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

10.4 Mappings

- Primarily `dict`: key/value store

10.5 Callables

- Functions, methods, classes, lambdas

---

11. Python Sequences

11.1 Indexing

```
s[i]
```

- Raises exception if out of range

11.2 Negative Indexing

```
s[-1]  # last element
```

11.3 Slicing

```
s[i:j]       # from i to j-1
s[:j]        # from beginning to j-1
s[i:]        # from i to end
s[:]         # copy entire list
```

11.4 Common Sequence Ops

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

11.5 Strings Are Immutable

Use slicing and string operations to manipulate.

---

12. Mutable Sequences: Lists

12.1 Assign to Element

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

12.2 Assign to Slice

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

Can grow or shrink list.

12.3 Delete Items

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

12.4 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

12.5 Efficiency of Append

- Python's `list.append()` is amortized O(1).
- Internally manages over-allocated capacity for efficiency.

Example performance analysis:

```
n total appends → total cost ≈ 2n → amortized O(1)
```

Allocates memory in powers of 2 to avoid frequent reallocation.

---

13. Summary

This lecture introduces Python from both basic and intermediate perspectives. It covers Python's core types and data structures, distinguishing between mutable and immutable objects, and demonstrates data parsing using list comprehensions and functions like `zip()`. The unique object model of Python is explained, illustrating identity, type, and value. Sequences such as strings and lists are reviewed in depth, including operations like indexing, slicing, and mutation. The efficiency of list operations such as `append()` is examined using amortized analysis.