

Python

Basics

Alish Bista - 10/10/2025

Language Semantics

Indentation

Between the Whitespaces

- Python uses whitespace (spaces/tabs) to organize code
- Colon (:) starts an indented block
- All code in a block must have same indentation
- Standard practice: use 4 spaces (not tabs!)
- Indentation shows which code belongs together

```
for x in [1, 2, 3]:  
    if x > 1:  
        print(f'{x} is  
greater than 1')  
  
    else:  
        print(f'{x} is  
not greater than 1')
```

Variables & Assignment

Assigning To Containers

- Variables created by assignment (using =)
- No need to declare type beforehand
- Variables are references to objects
- Multiple variables can point to same object
- Use meaningful names for variables

```
name = "Alice"  
age = 25  
height = 5.6  
  
# Multiple assignment  
a = b = [1, 2, 3]  
b.append(4)  
print(a) # Output: [1, 2,  
3, 4]
```

Everything is an Object

Objects Everywhere

- Numbers, strings, functions - all are objects
- Each object has a type (int, str, float, etc.)
- Use `type()` to check an object's type
- Use `isinstance()` to verify type
- Objects have methods and attributes

```
x = 5
print(type(x)) # <class
'int'>

y = "hello"
print(type(y)) # <class
'str'>
print(isinstance(y, str))
# True
```

Dynamic Typing

I Know Your Type

- Variables can change type through reassignment
- Python won't automatically convert between incompatible types
- This is called "strongly typed"
- Type errors help catch mistakes early
- Use explicit conversion when needed

```
a = 5          # a is integer
a = "hello"    # now a is
                string
                # This causes an error:
                # "5" + 5 # TypeError!
                # Correct way:
                "5" + str(5)      # "55"
                int("5") + 5     # 10
```

Scalar Types

Numbers

int and float

- **int**: whole numbers (can be arbitrarily large)
- **float**: decimal numbers (double precision)
- Division (/) always returns **float**
- Floor division (//) drops decimal part
- Use ** for exponentiation (power)

```
x = 10          # int  
y = 3.14        # float
```

```
print(7 / 2)    # 3.5  
(float division)  
  
print(7 // 2)   # 3 (floor  
division)  
  
print(2 ** 3)   # 8 (2 to  
the power of 3)
```

Strings

Text Handling

- Use single ' or double " quotes (both work the same)
- Triple quotes """ for multi-line strings
- Strings are immutable (cannot be changed)
- Use + to concatenate (join) strings
- Many useful methods available

```
name = 'Alice'  
greeting = "Hello"  
message = greeting + ", " +  
name # "Hello, Alice"  
  
# Multi-line string  
paragraph = """This is line 1  
This is line 2  
This is line 3"""
```

Strings

Methods

- `.upper()` and `.lower()` change case
- `.replace()` substitutes text
- `.split()` breaks string into list
- String slicing: `s[start:end]`
- Use f-strings for formatting

```
text = "python programming"  
print(text.upper())      #  
"PYTHON PROGRAMMING"  
  
print(text.replace("python",  
"Java"))    # "Java programming"  
  
# String slicing  
print(text[:6])      # "python"  
print(text[7:])    # "programming"
```

Strings

F-Strings

- Put f before the string
- Use {} to insert variables
- Can include expressions inside {}
- Can format numbers with : specifications
- Most readable way to format strings

```
name = "Alice"
age = 25
height = 5.6

# F-string formatting
print(f"My name is {name}")
print(f"I am {age} years old")
print(f"Height: {height:.1f} feet")

print(f"Next year I'll be {age + 1}")
```

Boolean

True False

- Two values: True and False (capital T and F!)
- Result from comparisons and conditions
- Combined with and, or, not keywords
- False, 0, None, empty collections are “falsy”
- Everything else is “truthy”

```
x = 5  
y = 10
```

```
print(x < y)           # True  
print(x == y)          # False  
print(True and False)  # False  
print(True or False)   # True  
print(not True)        # False
```

None

Nothing value

- None represents absence of value (like null)
- Only one None object exists
- Use is None to check for None
- Common default for function parameters
- Different from 0, False, or empty string

```
x = None  
print(x is None)           # True  
print(x is not None)      #  
False  
  
# Common pattern  
result = None  
if some_condition:  
    result = compute_value()
```

Type Conversion

Casting

- Convert between types using type functions
- `int()` converts to integer
- `float()` converts to decimal number
- `str()` converts to string
- `bool()` converts to True/False

```
x = "123"                                # 123
y = int(x)
(integer)

a = 5.7
b = int(a)                                # 5 (drops
decimal)

c = str(100)                               # "100"
(string)

print(bool(0))                            # False
print(bool(5))                            # True
```

Mutable / Immutable

Some change, Some don't

- **Immutable:** cannot be changed (str, int, float, tuple)
- **Mutable:** can be modified (list, dict, set)
- Immutable objects are safer and predictable
- Mutable objects are flexible but need care

```
# Immutable - creates new string
text = "hello"
text = text + " world" # new
string created

# Mutable - modifies in place
my_list = [1, 2, 3]
my_list.append(4)
# same list modified

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

Control Flow

If

If This Then That

- Check conditions and execute code accordingly
- Use elif for additional conditions
- Use else as fallback when all conditions are False
- Conditions must evaluate to True or False

```
temperature = 25  
  
if temperature > 30:  
    print("It's hot!")  
elif temperature > 20:  
    print("It's nice!")  
else:  
    print("It's cold!")
```

for

for looping over

- Repeat code for each item in a collection
- Works with lists, strings, ranges, etc.
- *continue* skips to next iteration
- *break* exits the loop early

```
fruits = ["apple", "banana",  
          "cherry"]
```

```
for fruit in fruits:  
    print(f"I like {fruit}")  
  
# Output:  
# I like apple  
# I like banana  
# I like cherry
```

while

While It Is True

- Repeat while condition is True
- Check condition before each iteration
- Use break to exit early
- Be careful of infinite loops!
- Good when you don't know iterations in advance

```
count = 0
while count < 5:
    print(f"Count is {count}")
    count += 1
```

range

In The Range

- range(n) generates 0 to n-1
- range(start, stop) generates start to stop-1
- range(start, stop, step) with custom increment
- Perfect for loops with indices

```
# 0 to 4
for i in range(5):
    print(i)
```

```
# 1 to 5
for i in range(1, 6):
    print(i)
```

```
# Even numbers 0 to 8
for i in range(0, 10, 2):
    print(i)
```

Modules

import

Importing Modules

- Modules are Python files with reusable code
- import brings in external functionality
- from module import name imports specific items
- import module as alias creates shorthand
- Expands Python's capabilities tremendously

```
# Import entire module  
import math  
print(math.sqrt(16)) # 4.0  
  
# Import specific function  
from math import sqrt  
print(sqrt(25)) # 5.0  
  
# Import with alias  
import math as m  
print(m.pi) # 3.14159...
```

datetime

Working with Dates and Time

- Use datetime module for date/time operations
- Create datetime objects with specific values
- Format dates using strftime()
- Parse strings using strptime()
- Calculate time differences easily

```
from datetime import datetime

# Create a datetime
dt = datetime(2024, 12, 25, 14, 30)
print(dt)    # 2024-12-25 14:30:00

# Current date and time
now = datetime.now()

# Format as string
print(now.strftime("%Y-%m-%d"))  # 2024-11-20
```

First Visualization

First Visualization

Not That Far!

- Python makes data analysis and visualization easy
- *dictionary* stores labeled data in key-value pairs: {'Product': 'Laptop', 'Sales': 120}
- *pandas* organizes data in tables (DataFrames)
- *seaborn* creates beautiful charts with just one line

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Create data using a dictionary
data = {
    'Product': ['Laptop', 'Phone', 'Tablet',
    'Watch', 'Headphones'],
    'Sales': [120, 200, 80, 150, 90]
}

# Convert to DataFrame (a table)
df = pd.DataFrame(data)
print(df)

sns.barplot(data=df, x='Product', y='Sales')
plt.title('Product Sales')
plt.show()
```

Q&A