Variables & Data Types in Python - Complete Notes
This PDF contains table format notes, interview questions, and practice problems on Python
variables and data types.
Data Type Description Example
int Whole numbers x = 10
float Decimal numbers pi = 3.14
bool True / False flag = True
str Text name = 'Aamir'
list Mutable sequence [1, 2, 3]
tuple Immutable sequence (1, 2, 3)
dict Key-value pairs {'a': 1}
set Unique unordered values {1, 2, 3}
NoneType Represents no value x = None
Interview Questions
1. What is a variable in Python?
2. What is dynamic typing?
3. Difference between mutable and immutable?
4. What are Python data types?
5. What is None in Python?
6. What is the difference between list and tuple?
7. How do you check the type of a variable?
8. What is a dictionary?
9. What is type casting?
10. Explain Python's memory model for variables.
Practice Problems
1. Create variables of all data types and print their types.
2. Convert a string '100' into int and float.
3. Write a program to swap two variables.
4. Create a list and modify its values. Can you modify a tuple?
5. Create a dictionary for a student with 3 key-value pairs.
6. Convert a list to a set. What changes?
7. Write code to check mutability of list vs tuple.
8. Create a complex number and print its real and imaginary parts.
9. Create a None variable and check its type.
10. Demonstrate dynamic typing with a variable. explain one by one every line and word

Variables & Data Types in Python - Complete Notes

Variables in Python

A **variable** is a named location in memory that stores a value. Think of it like a labeled container that holds data.

python

```python
x = 10
name = "Alice"
```

Data Types Table

Data Type	Description	Example
`int`	Whole numbers	`x = 10`
`float`	Decimal numbers	`pi = 3.14`
`bool`	True / False values	`flag = True`
`str`	Text (strings)	`name = 'Aamir'`
`list`	Mutable sequence	`[1, 2, 3]`
`tuple`	Immutable sequence	`(1, 2, 3)`
`dict`	Key-value pairs	`{'a': 1}`
`set`	Unique unordered values	`{1, 2, 3}`
`NoneType`	Represents no value	`x = None`

Interview Questions - Detailed Explanations

1. What is a variable in Python?

A variable is a symbolic name that refers to an object in memory. It's like a label attached to a value.

python

```
age = 25  # 'age' is a variable storing integer 25
```

2. What is dynamic typing?

Python determines the data type automatically at runtime. You don't need to declare types explicitly.

python

```python
x = 10      # x is integer
x = "hello" # Now x is string - allowed in Python
```

3. Difference between mutable and immutable?

- **Mutable**: Can be changed after creation (lists, dicts, sets)
- **Immutable**: Cannot be changed after creation (int, float, str, tuple)

python

```python
# Mutable example
list1 = [1, 2, 3]
list1[0] = 10  # Allowed

# Immutable example
tuple1 = (1, 2, 3)
# tuple1[0] = 10  # This would cause ERROR
```

4. What are Python data types?

Data types define the type of data a variable can hold and the operations that can be performed on it.

5. What is None in Python?

`None` represents the absence of a value. It's a singleton object of `NoneType` .

python

```python
result = None  # Variable with no value
```

6. Difference between list and tuple?

- **List**: Mutable, uses `[]`, can be modified
- **Tuple**: Immutable, uses `()`, cannot be modified

python

```python
my_list = [1, 2, 3]    # List - mutable
my_tuple = (1, 2, 3)   # Tuple - immutable
```

7. How do you check the type of a variable?

Use the `type()` function or `isinstance()` function.

python

```python
x = 10
print(type(x))        # <class 'int'>
print(isinstance(x, int))  # True
```

8. What is a dictionary?

A dictionary is an unordered collection of key-value pairs.

python

```python
student = {"name": "John", "age": 20, "grade": "A"}
```

9. What is type casting?

Converting one data type to another explicitly.

python

```python
x = "100"
x_int = int(x)     # Convert string to integer
x_float = float(x) # Convert string to float
```

10. Explain Python's memory model for variables

Python variables are references to objects in memory. Multiple variables can reference the same object.

```python
a = [1, 2, 3]
b = a  # Both a and b point to same list object
b[0] = 10
print(a)  # [10, 2, 3] - a is also changed
```

Practice Problems - Solutions with Explanations

1. Create variables of all data types and print their types

```python
# Integer
age = 25
print(f"age: {age}, type: {type(age)}")

# Float
height = 5.9
print(f"height: {height}, type: {type(height)}")

# Boolean
is_student = True
print(f"is_student: {is_student}, type: {type(is_student)}")

# String
name = "Alice"
print(f"name: {name}, type: {type(name)}")

# List
scores = [85, 90, 78]
print(f"scores: {scores}, type: {type(scores)}")

# Tuple
coordinates = (10, 20)
print(f"coordinates: {coordinates}, type: {type(coordinates)}")

# Dictionary
student = {"name": "Bob", "age": 22}
print(f"student: {student}, type: {type(student)}")
```

```python
# Set
unique_numbers = {1, 2, 2, 3, 3}
print(f"unique_numbers: {unique_numbers}, type: {type(unique_numbers)}")

# None
result = None
print(f"result: {result}, type: {type(result)}")
```

Explanation:

- Each variable is assigned a value of different data type
- `type()` function returns the class/type of the variable
- `f-string` formatting is used for clean output

2. Convert a string '100' into int and float

python

```python
# Original string
number_str = '100'
print(f"Original string: {number_str}, type: {type(number_str)}")

# Convert to integer
number_int = int(number_str)
print(f"After int conversion: {number_int}, type: {type(number_int)}")

# Convert to float
number_float = float(number_str)
print(f"After float conversion: {number_float}, type: {type(number_float)}")
```

Explanation:

- `int()` function converts string to integer
- `float()` function converts string to float
- The original string remains unchanged

3. Write a program to swap two variables

python

```python
# Method 1: Using temporary variable
a = 5
b = 10
print(f"Before swap: a = {a}, b = {b}")

temp = a
a = b
b = temp
print(f"After swap: a = {a}, b = {b}")

# Method 2: Pythonic way (tuple unpacking)
x = 15
y = 20
print(f"Before swap: x = {x}, y = {y}")

x, y = y, x
print(f"After swap: x = {x}, y = {y}")
```

Explanation:

- **Method 1**: Uses temporary variable to hold one value during swap
- **Method 2**: Python tuple unpacking - more elegant and efficient

4. Create a list and modify its values. Can you modify a tuple?

python

```python
# List - Mutable (can be modified)
fruits = ["apple", "banana", "cherry"]
print(f"Original list: {fruits}")

# Modify list
fruits[0] = "orange"  # Change first element
fruits.append("grape") # Add new element
print(f"Modified list: {fruits}")

# Tuple - Immutable (cannot be modified)
colors = ("red", "green", "blue")
print(f"Original tuple: {colors}")

# This would cause ERROR:
# colors[0] = "yellow"  # TypeError: 'tuple' object does not support item assignment

# To "modify" a tuple, create a new one
```

```python
new_colors = ("yellow",) + colors[1:]
print(f"New tuple: {new_colors}")
```

Explanation:

- Lists are **mutable** - elements can be changed, added, removed
- Tuples are **immutable** - cannot be changed after creation
- To "change" a tuple, create a new tuple with desired values

5. Create a dictionary for a student with 3 key-value pairs

python

```python
# Creating student dictionary
student = {
    "name": "Sarah",
    "age": 21,
    "major": "Computer Science"
}

print("Student Information:")
for key, value in student.items():
    print(f"{key}: {value}")

# Accessing values
print(f"\nStudent name: {student['name']}")
print(f"Student age: {student.get('age')}")

# Adding new key-value pair
student["gpa"] = 3.8
print(f"\nAfter adding GPA: {student}")
```

Explanation:

- Dictionary uses `{}` with `key: value` pairs
- Values can be accessed using `dict[key]` or `dict.get(key)`
- New key-value pairs can be added anytime

6. Convert a list to a set. What changes?

python

```python
# Original list with duplicates
numbers_list = [1, 2, 2, 3, 4, 4, 4, 5]
print(f"Original list: {numbers_list}")

# Convert to set
numbers_set = set(numbers_list)
print(f"After converting to set: {numbers_set}")

# Changes observed:
print("\nChanges observed:")
print(f"1. Duplicates removed: {numbers_set}")
print(f"2. Order may change (sets are unordered)")
print(f"3. Type changed from {type(numbers_list)} to {type(numbers_set)}")

# Demonstration of set operations
set_a = {1, 2, 3}
set_b = {3, 4, 5}
print(f"\nSet A: {set_a}")
print(f"Set B: {set_b}")
print(f"Union: {set_a | set_b}")
print(f"Intersection: {set_a & set_b}")
```

Explanation:

- **Duplicates removed**: Sets only store unique elements
- **Order not guaranteed**: Sets are unordered collections
- **Cannot access by index**: Use set operations instead

7. Write code to check mutability of list vs tuple

python

```python
def demonstrate_mutability():
    # List demonstration (Mutable)
    my_list = [1, 2, 3]
    print(f"Original list: {my_list}")
    print(f"List ID before modification: {id(my_list)}")

    # Modify the list
    my_list[0] = 10
    my_list.append(4)
    print(f"Modified list: {my_list}")
    print(f"List ID after modification: {id(my_list)}")
    print("List ID remains same - MUTABLE\n")
    # Tuple demonstration (Immutable)
    my_tuple = (1, 2, 3)
    print(f"Original tuple: {my_tuple}")
    print(f"Tuple ID: {id(my_tuple)}")

    # Try to modify tuple (this would cause error)
    try:
        my_tuple[0] = 10
    except TypeError as e:
        print(f"Cannot modify tuple: {e}")

    # Create new tuple
    new_tuple = my_tuple + (4,)
    print(f"New tuple: {new_tuple}")
    print(f"New tuple ID: {id(new_tuple)}")
    print("Different ID - IMMUTABLE (new object created)")

demonstrate_mutability()
```

Explanation:

- **List**: Same ID before/after modification - mutable
- **Tuple**: Cannot modify, new object created for changes - immutable
- `id()` function returns memory address of object

8. Create a complex number and print its real and imaginary parts

python

```python
# Creating complex numbers
complex_num1 = 3 + 4j
complex_num2 = complex(5, -2)

print(f"Complex number 1: {complex_num1}")
print(f"Complex number 2: {complex_num2}")

# Accessing real and imaginary parts
print(f"\nFor {complex_num1}:")
print(f"Real part: {complex_num1.real}")
print(f"Imaginary part: {complex_num1.imag}")

print(f"\nFor {complex_num2}:")
print(f"Real part: {complex_num2.real}")
print(f"Imaginary part: {complex_num2.imag}")
```

```
# Complex number operations
print(f"\nOperations:")
print(f"Addition: {complex_num1 + complex_num2}")
print(f"Multiplication: {complex_num1 * complex_num2}")
```

Explanation:

- Complex numbers use `j` for imaginary part
- `complex(real, imag)` constructor creates complex numbers
- `.real` and `.imag` attributes access parts
- Support mathematical operations

9. Create a None variable and check its type

python

```python
# Creating None variables
empty_value = None
result = None
default_value = None

print(f"empty_value: {empty_value}")
print(f"Type of empty_value: {type(empty_value)}")

# Practical use of None
def find_student(name, student_list):
    for student in student_list:
        if student["name"] == name:
            return student
    return None  # Not found

students = [{"name": "Alice", "age": 20}, {"name": "Bob", "age": 22}]

# Test the function
found = find_student("Alice", students)
print(f"\nFound Alice: {found}")

not_found = find_student("Charlie", students)
print(f"Found Charlie: {not_found}")
print(f"Type of not_found: {type(not_found)}")

# Checking for None
```

```python
if not_found is None:
    print("Student not found!")
```

Explanation:

- `None` represents absence of value
- Type is `NoneType`
- Commonly used as default return value
- Check with `is None` or `is not None`

10. Demonstrate dynamic typing with a variable

python

```python
print("=== DYNAMIC TYPING DEMONSTRATION ===\n")

# Single variable, multiple types
my_variable = 10
print(f"Stage 1 - Integer: {my_variable}, type: {type(my_variable)}")

my_variable = 3.14
print(f"Stage 2 - Float: {my_variable}, type: {type(my_variable)}")

my_variable = "Hello World"
print(f"Stage 3 - String: {my_variable}, type: {type(my_variable)}")

my_variable = [1, 2, 3]
print(f"Stage 4 - List: {my_variable}, type: {type(my_variable)}")

my_variable = {"key": "value"}
print(f"Stage 5 - Dictionary: {my_variable}, type: {type(my_variable)}")

my_variable = True
print(f"Stage 6 - Boolean: {my_variable}, type: {type(my_variable)}")

# Dynamic typing in action
print("\n=== PRACTICAL EXAMPLE ===")
data = 100
print(f"Data as integer: {data}, type: {type(data)}")

# Same variable can hold different types based on condition
if data > 50:
    data = "High value"
else:
    data = [1, 2, 3]

print(f"After condition, data: {data}, type: {type(data)}")

# Function demonstrating dynamic typing
def process_input(value):
    result = value * 2  # Operation depends on type of value
    print(f"Input: {value} (type: {type(value)}), Result: {result} (type: {type(resul
t)})")
    return result

print("\n=== SAME FUNCTION, DIFFERENT TYPES ===")
process_input(5)      # Integer input
process_input(2.5)    # Float input
process_input("Hi ")  # String input
process_input([1, 2]) # List input
```

Explanation:

- **Dynamic Typing**: Variable type determined at runtime
- **Same variable** can hold different data types
- **No type declarations** needed
- **Flexible** but requires careful type handling
- Python automatically manages type changes

Key Takeaways

1. **Variables** are references to objects in memory
2. **Dynamic typing** allows flexible variable usage
3. **Mutable** objects can be changed; **Immutable** cannot
4. **Type conversion** is done with `int()` , `float()` , `str()` , etc.
5. **None** represents absence of value
6. Use `type()` to check variable type
7. Lists are mutable, tuples are immutable