Console Input & Output in Python

Anab Batool Kazmi

Python Syntax

- Much of it is similar to C syntax
- Exceptions:
  - missing operators: `++`, `--`
  - no curly brackets, `{ }` , for blocks; uses whitespace
  - different keywords
  - lots of extra features
  - no type declarations!

Variables

- No need to declare
- Need to assign (initialize)
    - use of uninitialized variable raises exception
- Not typed

```
if friendly: greeting = "hello world"
else: greeting = 12**2
print greeting
```

- ***Everything*** is a variable:
    - functions, modules, classes

Basic Syntax Rules

- The name of your variable (**myInt** etc.) is placed on the left of the “=“ operator.
  - Most variable names are in **camel case** where the first word begins with a lowercase letter and any subsequent words are capitalized
  - Variable names may also appear in **snake case** where all words are lowercase, with underscores between words
- The assignment operator (**“=“**) sets the variable name equal to the memory location where your value is found.
- The value of your variable (**“Hello, World”**) is placed on the right of the “=“ operator.
  - The type of this value does **NOT** need to be stated but its format must abide by a given object type (as shown).

Reference semantics

- Assignment manipulates references
  - x = y **does not make a copy** of y
  - x = y makes x **reference** the object y references
- Very useful; but beware!
- Example:

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

Keywords, Identifiers and Literals

Keywords

- Keywords are reserved words in Python that have special meanings and specific purposes within the language.
- These keywords cannot be used as variable names, function names, or any other identifiers because they are already predefined in Python for particular tasks.
- Attempting to use a keyword as an identifier will result in a syntax error.

Python keyword module

- **Python** provides an in-built module called **keyword** that allows you to work with and access information about the reserved keywords in Python.
- The keyword module provides **functions and attributes** that can help you **identify and work** with Python's reserved keywords.

Python keyword module

Python Code

```
import keyword
print(keyword.kwlist)
```

Output

```
['False', 'None', 'True', 'and', 'as',
'assert', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except',
'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda',
'nonlocal', 'not', 'or', 'pass', 'raise',
'return', 'try', 'while', 'with', 'yield']
```

Python keyword module

Python Code

```
import keyword


word = "if"
is_keyword = keyword.iskeyword(word)
print(f"'{word}' is a keyword: {is_keyword}")
```

Output

```
'if' is a keyword: True
```

Identifiers

- Identifiers are **names** given to various program elements such as **variables, classes, functions, lists, methods, and more**.
- They are used to uniquely recognize and refer to these elements within the code.

Rules for Naming Python Identifiers

1. **Not a Reserved Keyword:** An identifier cannot be a reserved Python keyword, as these words have predefined meanings in the language.
2. **No White Spaces:** Identifiers should not contain white spaces or any form of whitespace.
3. **Character Composition:** Identifiers can consist of a combination of uppercase letters (A-Z), lowercase letters (a-z), digits (0-9), or underscores (_).
4. **Starts with Letter or Underscore:** An identifier must start with an alphabetic character (A-Z or a-z) or an underscore (_).
5. **No Special Characters:** Identifiers should not contain special characters or symbols other than the underscore (_).

Literals

- Literals in programming are indeed ***a notation used to represent fixed or constant values.***
- Literals can represent values of various data types, such as integers, floating-point numbers, strings, characters, and more

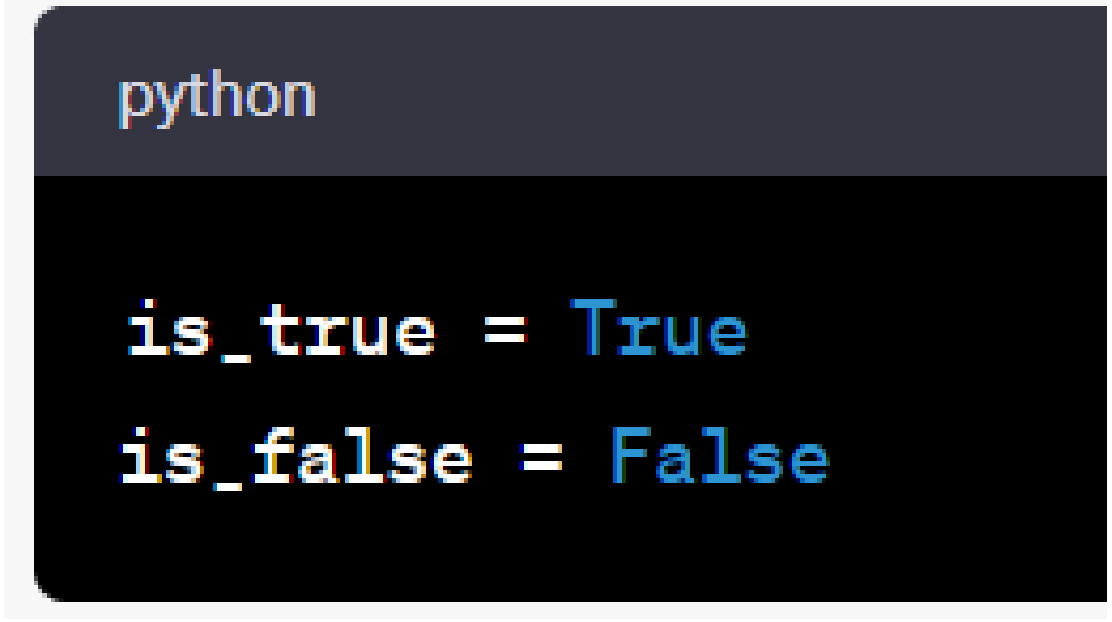
- For example:
- 42 is a literal representing an integer value.
- 3.14 is a literal representing a floating-point value.
- "Hello, World!" is a literal representing a string value.
- 'A' is a literal representing a character value.

Types of Literals in Python

- Boolean literals
- Numeric literals
- Character literals
- String literals
- Literal Collections
- Special literals

Boolean Literals

- Boolean literals represent two constant values: True and False.
- They are used to represent the truth values in logical operations.



Numeric Literals

- Numeric literals are used to represent numeric values.
- There are different types of numeric literals, including integers (binary, decimal, octal, hexadecimal), floating-point numbers, and complex numbers.

```python
integer_literal = 42
float_literal = 3.14
complex_literal = 2 + 3j
```

Character Literal

- Python doesn't have a distinct character data type, so **single-character strings** can be considered character literals.
- They are enclosed in **single quotes or double quotes.**.

```python
char_literal = 'A'
```

String literals

- String literals are used to represent text or sequences of characters.
- They are enclosed in single (' '), double (" "), or triple (''' ' ''', """ """ or ''' ''') quotes.

python

```python
single_quoted = 'Hello, World!'
double_quoted = "Python is great!"
triple_quoted = '''This is a
multiline string.'''
```

Literal Collections

- Python provides literal notations for collections like **lists, tuples, dictionaries, and sets.**

```python
list_literal = [1, 2, 3]
tuple_literal = (4, 5, 6)
dict_literal = {'name': 'Alice', 'age': 30}
set_literal = {1, 2, 3}
```

Special literals

- python has two special literals: **None** and **Ellipsis**.
- **None** represents the **absence of a value or a null value**.
- **Ellipsis** (or ...) is used in slicing to indicate a **range or continuation**.

python

```python
empty_value = None
sliced_data = data[2:5]  # Ellipsis indicates a range
```

Input and Output in Python

Input in Python

- Input in Python is the process of **receiving data or values from external sources, such as the user, files, or other programs.**
- The primary way to receive user input in Python is by using the **input() function**.
- **It reads a line of text entered by the user and returns it as a string by default.**

input() Function

- The input() function is used to ***read a line of text entered by the user*** from the standard input, which is typically ***the console or terminal*** where a user interacts with the program by typing text.
- It takes an **optional prompt (a string) as an argument**, which is **displayed to the user** to indicate what input is expected.
- The user enters data and presses the Enter key.
- The input() function **reads the entered text as a string** and returns it.

python

```python
user_input = input("Enter something: ")
```

Output in Python

- Output in Python refers to the process of **displaying or presenting data, messages, or results to the user**, typically through the standard output, which is often the **console or terminal**.
- The primary way to produce output in Python is by using the **print() function.**

```
print(object(s), sep=' ', end='\n', file=file, flush=False)
```

Multiple Arguments

- You can pass multiple arguments to print(), and it will print each argument separated by a space by default.

```
1  #Multiple Arguments
2  name = "Alice"
3  age = 30
4  print("Name:", name, "Age:", age)  # Display multiple variables
5
```

```
Name: Alice Age: 30
```

Formatted Output

- You can format the output using placeholders and the format() method or f-strings (Python 3.6+).

```python
#Formatted Output
name = "Alice"
age = 30
print("Name: {}, Age: {}".format(name, age))  # Using format()

# Using f-strings (Python 3.6+)
print(f"Name: {name}, Age: {age}")
```

```
Name: Alice, Age: 30
Name: Alice, Age: 30
```

```
print(object(s), sep=' ', end='\n', file=file, flush=False)
```

- **"object(s)"** represent values for display.
- They are **automatically converted to strings** before printing.
- This primary argument allows **passing one or more objects**, with the **sep** parameter specifying the separator.

```
1 print("1. hello world!") # Display a string literal
2 print("2. BSAI 4!" , "welcome to the class") # Display multiple messages by usng default seperato
3 print(12345) # Display a integer literal
4 print(False) # Display a boolean literal
5 print(0b10100, 50, 0o320, 0x12b) # deciman conversion and print
```

```
1. hello world!
2. BSAI 4! welcome to the class
12345
False
20 50 208 299
```

```
print(object(s), sep=' ', end='\n', file=file, flush=False)
```

- The **sep** parameter is an **optional parameter** that specifies the **separator to be used between multiple objects** passed to the print() function.
- **By default, it is set to a space character (' '),** meaning that objects will be separated by spaces. You can change this separator by providing a different string.

```
1 print(" hello world!", 1234) #Printing Multiple Objects with Default Separator
2 print("hello world!", 1234, sep='$') #Custom Separator and End Character
3 # printing variable values
4 name = "Alice"
5 age = 30
6 print(name, age)  #Printing Multiple Objects with Default Separator
7 print(name, age, sep=" | ") #Custom Separator
```

```
 hello world! 1234
hello world!$1234
Alice 30
Alice | 30
```

```
print(object(s), sep=' ', end='\n', file=file, flush=False)
```

- The **end** parameter is another optional parameter that determines what character(s) will be printed **at the end of the print() function's output**.
- By default, it is set to a newline character ('\n'), which means that each print() call ends with a newline, creating a new line for the next output.
- You can change this character to something else if needed.

```
1 print(name, age, sep=" | ", end=" | ") #Custom Separator and end
2 print("hello world!")# this will get printed with the previous line
```

```
Alice | 30 | hello world!
```

```
print(object(s), sep=' ', end='\n', file=file, flush=False)
```

- The **file** parameter is optional and allows you to specify a file-like object to which the output will be written.
- By default, it is set to **None**, meaning that the output will be displayed in the console.
- If you want to redirect the output to a file, you can provide the file object here.

```
# Redirecting output to a file
with open("output.txt", "w") as file:
    print("This will be written to a file.", file=file)
```

```
print(object(s), sep=' ', end='\n', file=file, flush=False)
```

- The **flush parameter** is optional and determines whether the output should be flushed (written immediately) or buffered (saved in memory until a certain condition is met).
- By default, it is set to False, meaning that output is buffered. Setting it to True forces the output to be flushed immediately.

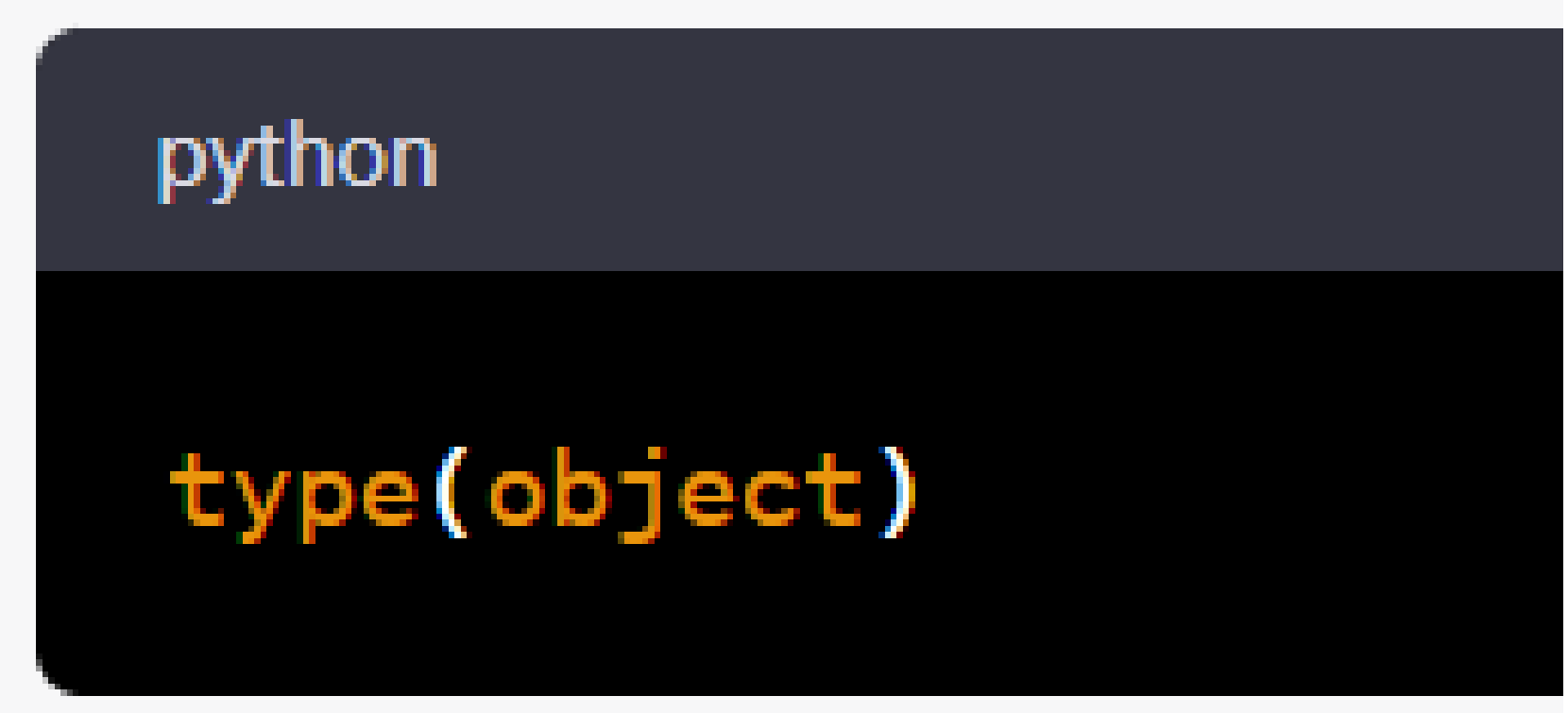
Type casting in Python

Type casting

- Type Casting, also known as **Type Conversion**, is the process of **converting the data type of a variable or value from one type to another.**
- Python supports two main types of Type Casting:
  - Python Implicit Type Conversion (Automatic Type Conversion)
  - Python Explicit Type Conversion (Manual Type Conversion)

type() function

- the **type() function** is used to determine the **data type of an object or a value**. It **returns the class type** of the argument(s) passed to it.



type() function

Python Code

```
1  a=5
2  b=0o320
3  c=5.5
4  d=2+3j
5  e='a'
6  f='apple'
7  g=[a,b,c]
8
9  print(type(a))
10 print(type(b))
11 print(type(c))
12 print(type(d))
13 print(type(e))
14 print(type(f))
15 print(type(g))
16
```

Output

```
<class 'int'>
<class 'int'>
<class 'float'>
<class 'complex'>
<class 'str'>
<class 'str'>
<class 'list'>
```

Python Implicit Type Conversion

- This type of conversion is **done automatically by Python** when an operation between different data types is performed, and it doesn't result in any data loss or error.

```
1  #implicit type casting
2  int_num = 10
3  float_num = 5.5
4
5  result = int_num + float_num  # Python implicitly converts int to float for the addition
6  print(type(result))
7  print(result)
8
9
```

```
<class 'float'>
15.5
```

Python Explicit Type Conversion

- This type of conversion is performed **manually** by the programmer using **built-in functions or constructors** to explicitly change the data type of a variable.
- Explicit Type Conversion is necessary when there's a possibility of data loss or when you want to ensure the data is of a specific type.
- Common functions used for explicit type conversion include int(), float(), str(), bool() and others.

```
1  #Explicitly converting a string to an integer
2  number = int("56")
3  print(type(number))
4  print(number)
5
```

```
<class 'int'>
56
```

Python Explicit Type Conversion

```
1  name_str = input("Enter your name: ")
2  age_str = input("Enter your age: ")
3  age = int(age_str)  # Convert the string to an integer
4  gpa_str = float(input("Enter your gpa: ")) # Convert the string to float
```

```
Enter your name: Alice
Enter your age: 30
Enter your gpa: 2.4
```

Python Explicit Type Conversion

- **int(x, base=10):** Converts x to an integer. The optional base parameter specifies the base of the number if x is a string.
- **float(x):** Converts x to a floating-point number.
- **str(x):** Converts x to a string representation.
- **bool(x):** Converts x to a Boolean value (True or False).
- **list(iterable):** Converts an iterable (e.g., a tuple, set, or string) to a list.

Python Explicit Type Conversion

- **tuple(iterable):** Converts an iterable to a tuple.
- **set(iterable):** Converts an iterable to a set.
- **dict(iterable):** Converts an iterable of key-value pairs to a dictionary.
- complex(real, imag): Creates a complex number with the given real and imaginary parts.
- .
- .
- .

Any Question So Far?