# hello, data type, id, size, variable, type cast

print("hello")

var1=10  
var2=10.10  
var3=12j  
var4=12+12j  
print(type(var1))  
print(type(var2))  
print(type(var3))  
print(type(var4))  
  
# <class 'int'>  
# <class 'float'>  
# <class 'complex'>  
# <class 'complex'>

var=100  
print(id(var))  
# 140736988581896  
  
# id(var) returns the memory address (or a unique identifier) where the integer object 100 is stored.  
# This value is typically represented as a large integer (like 140736988581896 in your case).  
# The exact value will differ between different Python runs or systems.

from sys import getsizeof  
  
var1=100  
import sys  
print(sys.getsizeof(var1))  
# 28  
# will output the size (in bytes) of the integer object var1 using Python’s sys.getsizeof() function.  
# var1 = 100: Assigns the integer value 100 to the variable var1.  
# import sys: Imports the sys module.  
# sys.getsizeof(var1): Returns the size in bytes of the object var1, which includes the overhead for Python’s object management.  
  
  
var="hello python"  
print(getsizeof(var))  
# 53

# var =10  
# print(var)  
  
# var=10  
# var=100  
var=1000  
print(var)  
# last value will be print  
  
var1=100  
var2="hello"  
var3=200  
# print("hello" + str(var3))  
# print("hello",var3)  
print(f"hello {var3}")  
# NB:  
# Option 1: Convert the number to a string  
# Option 2: Use commas in print (which automatically adds spaces)  
# Option 3: Use f-strings (Python 3.6+)

x = 10  
print(type(x))  
x1 = float(x)  
print(x1)  
print(type(x1))  
x2 = complex(x1)  
print(x2)  
print(type(x2))  
  
# <class 'int'>  
# 10.0  
# <class 'float'>  
# (10+0j)  
# <class 'complex'>  
  
print("next")  
a = 10  
b = 20  
c = complex(a, b)  
print(c)  
# (10+20j)  
d = complex(b, a)  
print(d)  
# (20+10j)

Here's a summary of the provided code:

1. **Printing a String**:
   * print("hello"): Displays the string "hello".
2. **Variable Types**:
   * var1 = 10 (int), var2 = 10.10 (float), var3 = 12j (complex), var4 = 12+12j (complex).
   * type() function is used to check the type of each variable, which returns:
     + <class 'int'>
     + <class 'float'>
     + <class 'complex'>
3. **Memory Address**:
   * id(var) gives the unique identifier (memory address) where the variable is stored. Example: id(var) = 140736988581896.
4. **Size of Objects**:
   * sys.getsizeof(var1) shows the memory size of an object in bytes. Example: sys.getsizeof(var1) for 100 is 28 bytes.
5. **String Manipulation**:
   * To print a number alongside a string, you can:
     + Convert the number to a string and concatenate.
     + Use commas in print(), which adds spaces automatically.
     + Use f-strings (e.g., f"hello {var3}").
6. **Type Conversion**:
   * Converting x = 10:
     + From int to float: x1 = float(x) (10.0).
     + From float to complex: x2 = complex(x1) (10+0j).
     + You can convert numbers into different types (int, float, complex) as needed.
7. **Creating Complex Numbers**:
   * c = complex(a, b) creates a complex number with real part a and imaginary part b (e.g., (10+20j)).
   * You can reverse the real and imaginary parts, like d = complex(b, a) which gives (20+10j).

This covers basic operations like checking types, memory usage, size of objects, string formatting, type conversion, and complex number creation.

# Boolean, List, Tuple, Set, String, Range, and Dictionary

# boolean data type  
x = 10  
y = 20  
print(x > y)  
# False  
print(y > x)  
# True  
  
# list data type  
lst = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]  
print(lst)  
print(type(lst))  
# [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]  
# <class 'list'>  
  
  
# tuple data type  
tpl = (10, 20, 30, 40, 50, 60, 70, 80, 90, 100)  
print(tpl)  
print(type(tpl))  
# (10, 20, 30, 40, 50, 60, 70, 80, 90, 100)  
# <class 'tuple'>  
  
# set data type  
st = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}  
print(st)  
print(type(st))  
# {100, 70, 40, 10, 80, 50, 20, 90, 60, 30}  
# <class 'set'>  
  
# string data type  
st = 'python'  
print(st)  
print(type(st))  
# python  
# <class 'str'>  
  
# range data type  
rng = range(10)  
print(rng)  
print(list(rng))  
# range(0, 10)  
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
# range(10) creates a range object from 0 to 9.  
# Printing it directly shows range(0, 10); use list(range(10)) to see the numbers.  
  
# dictionary  
dct = {'name': 'mahade', 'email': 'wtbl.hasan@gmail.com', 'contact': '01771752777'}  
print(dct)  
# {'name': 'mahade', 'email': 'wtbl.hasan@gmail.com', 'contact': '01771752777'}  
print(type(dct))  
# <class 'dict'>  
print(dct.keys())  
# dict\_keys(['name', 'email', 'contact'])  
print(dct.values())  
# dict\_values(['mahade', 'wtbl.hasan@gmail.com', '01771752777'])

Here are the core data types used in the provided code:

1. **Boolean** (bool)  
   Example: x = 10, y = 20  
   Operations like comparison (x > y) are performed to return True or False.
2. **List** (list)  
   Example: lst = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]  
   A list is an ordered collection of items, and the type() function is used to check its type.
3. **Tuple** (tuple)  
   Example: tpl = (10, 20, 30, 40, 50, 60, 70, 80, 90, 100)  
   Tuples are similar to lists, but they are immutable (cannot be changed).
4. **Set** (set)  
   Example: st = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}  
   A set is an unordered collection with no duplicate elements.
5. **String** (str)  
   Example: st = 'python'  
   A string is a sequence of characters enclosed in quotes.
6. **Range** (range)  
   Example: rng = range(10)  
   A range is a sequence of numbers, often used in loops.
7. **Dictionary** (dict)  
   Example: dct = {'name': 'mahade', 'email': 'wtbl.hasan@gmail.com', 'contact': '01771752777'}  
   A dictionary is a collection of key-value pairs.

Each of these data types is fundamental to Python programming and showcases the variety of ways to store and manipulate data.

Here's a simple and clear breakdown of the **fundamental differences** among **List**, **Tuple**, and **Set** in Python:

| **Feature** | **List** | **Tuple** | **Set** |
| --- | --- | --- | --- |
| **Syntax** | [1, 2, 3] | (1, 2, 3) | {1, 2, 3} |
| **Mutable?** | ✅ Yes | ❌ No (Immutable) | ✅ Yes (but elements must be immutable) |
| **Ordered?** | ✅ Yes (insertion order) | ✅ Yes (insertion order) | ❌ No |
| **Allows Duplicates?** | ✅ Yes | ✅ Yes | ❌ No (only unique values) |
| **Indexable?** | ✅ Yes | ✅ Yes | ❌ No (not indexable) |
| **Use Case** | General-purpose collection with order and duplicates | Fixed data structure, faster access, hashable | Unique items, fast membership testing |

**Example:**

# List

my\_list = [1, 2, 3, 1]

my\_list[0] = 10 # Can change items

# Tuple

my\_tuple = (1, 2, 3)

# my\_tuple[0] = 10 # ❌ Error - can't change

# Set

my\_set = {1, 2, 3, 1}

print(my\_set) # {1, 2, 3} - duplicates removed

Let me know if you'd like visual examples or deeper use cases.

**Mutable means:**

✅ **Can be changed or modified after creation**

**Immutable means:**

❌ **Cannot be changed once created**

**🔄 Example of Mutable (List):**

my\_list = [1, 2, 3]

my\_list[0] = 10 # ✅ You can change elements

my\_list.append(4) # ✅ You can add new items

print(my\_list) # Output: [10, 2, 3, 4]

**🚫 Example of Immutable (Tuple):**

my\_tuple = (1, 2, 3)

# my\_tuple[0] = 10 ❌ This will give an error

# Tuples can't be changed once created

**📌 Quick Tip:**

* Mutable objects: list, set, dict
* Immutable objects: tuple, str, int, float

1. **Arithmetic** (+, -, /, %)
2. **Comparison** (==, !=, >, <)
3. **Increment** (+=, -=)
4. **Negation** (Unary -)
5. **Logic** (and, or, not)
6. **Conversion** (bin(), oct(), hex())
7. **Swapping** (Variable exchange)

x = 100  
y = 200  
print(x + y)  
# 300  
  
x1 = 100  
x2 = 10  
print(x1 / x2)  
print(x1 % x2)  
# 10.0  
# 0  
  
x1 = 10  
x2 = 20  
print(x1 == x2)  
# False  
  
print(x1 > x2)  
# False  
  
print(x1 != x2)  
# True  
  
**# increment decrement**  
x3 = 10  
x3 = x3 + 10  
print(x3)  
  
x3 += 50  
print(x3)  
# 70  
  
x3 -= 20  
print(x3)  
# 50  
  
**# unary operator**  
x4 = 10  
y4 = -x4  
print(y4)  
  
**# relational operator**  
a = 200  
b = 400  
c = a < b  
print(c)  
# True  
c = a != b  
print(c)  
# True

**# logical operator**  
a = 10  
b = 20  
# c = a < b and b < 5  
c = a < b < 5  
print(c)  
# False  
  
a = True  
b = not a  
print(b)  
# False  
 **# conversion  
# decimal to binary**  
a = 55  
b = bin(a)  
print(b)  
# 0b11001  
  
**# decimal to octal**  
x = 45  
y = oct(x)  
print(y)  
# 0o55

**# decimal to hex**  
x = 45  
y = hex(x)  
print(y)  
# 0x2d  
  
  
**# binary to decimal**  
def bin\_to\_dec(str):  
 dec = int(str, 2)  
 print(dec)  
  
binStr = "110111"  
bin\_to\_dec(binStr)  
# 55  
  
**# swapping**  
value1 = 100  
value2 = 200  
temp = value2  
value2 = value1  
value1 = temp  
print('value1=', value1)  
print('value2=', value2)  
# value1= 200  
# value2= 100  
  
**# using bitwise XOR Operation**  
a = 5000  
b = 10000  
a = a ^ b  
b = b ^ a  
a = a ^ b  
print('a=', a)  
print('b=', b)  
# a= 10000  
# b= 5000  
  
**# more simplified ways**  
a, b = b, a  
print(a, b)  
# 5000 10000

**Core Concepts Summary**

**1. Basic Arithmetic Operations**

* + (Addition), - (Subtraction), / (Division), % (Modulus/Remainder)
* Example:
  + 100 + 200 = 300
  + 100 / 10 = 10.0
  + 100 % 10 = 0

**2. Comparison Operators**

* == (Equal), != (Not Equal), > (Greater Than), < (Less Than)
* Example:
  + 10 == 20 → False
  + 10 != 20 → True
  + 200 < 400 → True

**3. Increment & Decrement**

* += (Add and assign), -= (Subtract and assign)
* Example:
  + x3 = 10 → x3 += 50 → x3 = 60
  + x3 -= 20 → x3 = 40

**4. Unary Operator (**-**)**

* Negates a value.
* Example:
  + y4 = -x4 (if x4 = 10, then y4 = -10)

**5. Logical Operators**

* and, or, not
* Example:
  + a < b and b < 5 → False
  + not True → False

**6. Number System Conversions**

* **Decimal to Binary (**bin()**)**: bin(55) → '0b110111'
* **Decimal to Octal (**oct()**)**: oct(45) → '0o55'
* **Decimal to Hexadecimal (**hex()**)**: hex(45) → '0x2d'
* **Binary to Decimal (**int(str, 2)**)**
  + int("110111", 2) → 55

**7. Swapping Variables**

* **Using Temporary Variable:**

temp = a

a = b

b = temp

* **Using XOR (Bitwise):**y

a = a ^ b

b = b ^ a

a = a ^ b

* **Simplest Way (Pythonic):**hon

a, b = b, a

**Key Takeaways**

1. **Arithmetic & comparison operators** are fundamental for calculations and conditions.
2. **Shortcut assignments (**+=**,**-=**)** simplify variable updates.
3. **Logical operators** (and, or, not) control program flow.
4. **Number conversions** (bin, oct, hex) help in different numeral systems.
5. **Swapping variables** can be done in multiple ways (temp variable, XOR, Python tuple swap).

# sqrt, ceil, floor, pow

import math  
x = **math.sqrt**(9)  
print(x)

import math  
x = math.sqrt(10)  
print(x)  
# 3.1622776601683795  
print(**math.ceil(x))**  
# 4  
print(**math.floor(x))**  
# 3  
print(**math.pow(3, 2))**  
# 9.0

**x = int(input('x='))  
y = int(input('y='))**  
print(x + y)  
name = input('name=')  
print(name)

Here are the results for the core functions with the smallest examples:

1. **math.sqrt(9)**: The square root of 9 is 3.0.
2. **math.ceil(3.2)**: The ceiling of 3.2 (rounding up) is 4.
3. **math.floor(3.8)**: The floor of 3.8 (rounding down) is 3.
4. **math.pow(3, 2)**: 3 raised to the power of 2 is 9.0. ​

**slicing syntax in Python**

# sequence[start:stop:step] # start: index to start (inclusive) # stop: index to stop (exclusive)  
# step: how many steps to move (default is 1)  
  
name = "mahade"  
print(name[:-1])  
# mahad  
  
name = "mahade"  
print(name[:-2])  
# maha  
  
name = "mahade"  
print(name[-4:])  
# hade  
  
name = "mahade"  
print(name[-4:-1])  
# had  
  
print(name[::-1])  
# edaham  
  
print(name[::2])  
# mhd

Here's a brief analysis of each slicing operation:

1. **name[:-1]**:
   * Slices the string from the beginning to the second-to-last character.
   * Result: "mahad"
2. **name[:-2]**:
   * Slices the string from the beginning to the third-to-last character.
   * Result: "maha"
3. **name[-4:]**:
   * Starts from the fourth-to-last character and slices to the end of the string.
   * Result: "hade"
4. **name[-4:-1]**:
   * Starts from the fourth-to-last character and ends just before the last character.
   * Result: "had"
5. **name[::-1]**:
   * Reverses the string.
   * Result: "edaham"
6. **name[::2]**:
   * Selects every second character from the string, starting from the first character.
   * Result: "mhd"

# While loop

# condition elif while  
age = int(input('x='))  
if age < 30:  
 print('less than 30')  
elif age <= 40:  
 print('less than or equal 40')  
else:  
 print('greater than 40')  
  
# x=40  
# less than or equal 40  
  
x = int(input('x='))  
if x == 1:  
 print('one')  
elif x == 2:  
 print('two')  
elif x == 3:  
 print('Three')  
elif x == 4:  
 print('Four')  
else:  
 print('Other Value')  
  
**i = 1  
while i <= 10:  
 print(i)  
 i += 1**  
password = input('password: ')  
while password != 'secret':  
 print('password not matched')  
 password = input('password: ')  
 if password == 'secret':  
 print('login successful')  
  
**x = int(input('x='))  
y = 'mahade'  
z = 0**  
  
**while z < x:  
 print(y)  
 z = z + 1**  
  
# x=5  
# mahade  
# mahade  
# mahade  
# mahade  
# mahade  
  
**# break**  
**while z < x:  
 print(y)  
 z = z + 1  
 if z == 3:  
 break**  
# x=10  
# mahade  
# mahade  
# mahade  
  
**# continue**  
**x = int(input('x='))  
z = 0  
while z < x:  
 z = z + 1  
 if z == 3:  
 continue  
 print(z)**

Sure! Let's break it down with simple explanations and examples of how break and continue work inside a while loop in Python.

**🔁 while loop in Python**

A while loop keeps running **as long as a condition is True**.

**✅ break in a while loop**

* **Purpose:** It **immediately exits** the loop, even if the condition is still true.
* **Use case:** When you want to **stop the loop early** under certain conditions.

**Example:**

i = 1

while i <= 10:

print(i)

if i == 5:

break # exits the loop when i is 5

i += 1

🟩 **Output:**

1

2

3

4

5

See? It stops as soon as i == 5.

**✅ continue in a while loop**

* **Purpose:** It **skips the rest of the current loop** iteration and moves to the **next iteration**.
* **Use case:** When you want to **ignore** specific cases.

**Example:**

i = 0

while i < 10:

i += 1

if i == 5:

continue # skips printing when i is 5

print(i)

🟨 **Output:**

1

2

3

4

6

7

8

9

10

See how 5 is missing? Because continue skipped it.

**Summary Table:**

| **Keyword** | **Action** |
| --- | --- |
| break | Exits the loop completely |
| continue | Skips the current iteration only |

Here's a quick summary of how to **initialize** each basic data type in Python and the brackets they use:

| **Data Type** | **Name** | **Example Initialization** | **Brackets Used** | **Notes** |
| --- | --- | --- | --- | --- |
| list | List | [] or list() | Square brackets [] | Mutable, ordered, allows duplicates |
| tuple | Tuple | () or tuple() | Parentheses () | Immutable, ordered, allows duplicates |
| set | Set | set() | No literal for empty set! Use curly braces {} for non-empty | Unordered, no duplicates |
| dict | Dictionary | {} or dict() | Curly braces {} | Key-value pairs, mutable, unordered (Python 3.7+ preserves ins |

**# List**

my\_list = [1, 2, 3]

**# Tuple**

my\_tuple = (1, 2, 3)

**# Set**

my\_set = {1, 2, 3}

empty\_set = set() # NOT {} – that's a dict

**# Dictionary**

my\_dict = {"a": 1, "b": 2}

**print('for-list')**

lst = ['mahade', 10, True, False, 100]

lst[0] = 'Afia'

print(lst)

print(lst[2])

for value in lst:

print(value)

**print('for-tuple')**

tpl = (10, True, False, 'Afia')

print(tpl[0])

for value in tpl:

print(value)

**print("for-set")**

st = {10, 20, 'mahade', 'hasan', 10}

# {10, 'mahade', 20, 'hasan'}

print(st)

for value in st:

print(value)

**print('break with for loop')**

lst = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

for item in lst:

print(item)

if item == 40:

break

for item in lst:

if item == 40:

continue

print(item)

for i in range(10):

print(i)

for i in range(10, 100, 13):

print(i)

# for-list

# ['Afia', 10, True, False, 100]

# True

# Afia

# 10

# True

# False

# 100

# for-tuple

# 10

# 10

# True

# False

# Afia

# for-set

# {'hasan', 10, 20, 'mahade'}

# hasan

# 10

# 20

# mahade

# break with for loop

# 10

# 20

# 30

# 40

# 10

# 20

# 30

# 50

# 60

# 70

# 80

# 90

# 100

# 0

# 1

# 2

# 3

# 4

# 5

# 6

# 7

# 8

# 9

# 10

# 23

# 36

# 49

# 62

# 75

# 88

for num in range(10, 30):

if num % 2 == 0:

print(num)

# 10

# 12

# 14

# 16

# 18

# 20

# 22

# 24

# 26

# 28

for num in range(10, 30):

if num % 2 != 0:

print(num)

# 13

# 15

# 17

# 19

# 21

# 23

# 25

# 27

# 29

**# tuple**

print('tuple')

tpl = (10, 20, 'mahade', 30, True, False, 50, 60, 70, 80, 90, 100)

print(tpl)

**print('tuple-break')**

for value in tpl:

print(value)

if value == 20:

break

**print('tuple-continue')**

for value in tpl:

if value == 'mahade':

continue

print(value)

# tuple

# (10, 20, 'mahade', 30, True, False, 50, 60, 70, 80, 90, 100)

# tuple-break

# 10

# 20

# tuple-continue

# 10

# 20

# 30

# True

# False

# 50

# 60

# 70

# 80

# 90

# 100

**# list comprehension**

print('list-comprehension')

lst = [num for num in range(20)]

print(lst)

# list-comprehension

# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

# num is the loop variable that takes values from range(20).

# So inside the list comprehension, you should use num, not item.

**lst = [num for num in range(16) if num % 2 == 0]**

print(lst)

# [0, 2, 4, 6, 8, 10, 12, 14]

**lst = [num for num in range(16) if num % 2 != 0]**

print(lst)

# [1, 3, 5, 7, 9, 11, 13, 15]

**print('list append')**

lst = []

for i in range(10):

lst.append(i)

print(lst)

# list append

# [0]

# [0, 1]

# [0, 1, 2]

# [0, 1, 2, 3]

# [0, 1, 2, 3, 4]

# [0, 1, 2, 3, 4, 5]

# [0, 1, 2, 3, 4, 5, 6]

# [0, 1, 2, 3, 4, 5, 6, 7]

# [0, 1, 2, 3, 4, 5, 6, 7, 8]

# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

**# tpl = ()**

# for i in range(5):

# tpl.append(i)

# print(tpl)

# You're trying to append elements to a tuple in Python, but tuples are immutable, which means you cannot change them (e.g., add items) after they are created.

**st = set()**

**for i in range(5):**

st.add(i)

print(st)

# {0}

# {0, 1}

# {0, 1, 2}

# {0, 1, 2, 3}

# {0, 1, 2, 3, 4}

**lst = [x for x in range(10) if x % 2 == 0]**

print(lst)

# [0, 2, 4, 6, 8]

**lst = []**

for x in range(10):

if x % 2 != 0:

lst.append(x)

print(lst)

# [1]

# [1, 3]

# [1, 3, 5]

# [1, 3, 5, 7]

# [1, 3, 5, 7, 9]

Here’s a **summary** of the provided Python code, broken down into logical sections:

**🔁 Working with Lists, Tuples, and Sets**

* **List (lst)**: Demonstrates list creation, element update, indexing, and iteration using a for loop.
* **Tuple (tpl)**: Shows tuple creation, indexing, and iteration. Tuples are immutable (cannot be modified after creation).
* **Set (st)**: Shows set creation (automatically removes duplicates) and iteration. Order is not guaranteed.

**🔄 Loop Control Statements**

* **break**: Stops the loop when a condition is met (item == 40).
* **continue**: Skips the current iteration when a condition is met (item == 40).
* **range()**:
  + range(10) – loops from 0 to 9.
  + range(10, 100, 13) – loops from 10 to 99, stepping by 13.

**🔢 Even/Odd Number Filtering**

* **Even numbers** in range(10, 30): if num % 2 == 0
* **Odd numbers** in range(10, 30): if num % 2 != 0

**🧵 Tuple Loops with break and continue**

* Breaks the loop when value == 20.
* Skips 'mahade' during iteration.

**⚙️ List Comprehension**

* Creates lists in a concise way.
  + [num for num in range(20)] – all numbers 0–19
  + [num for num in range(16) if num % 2 == 0] – even numbers
  + [num for num in range(16) if num % 2 != 0] – odd numbers

**📥 Appending to List and Set**

* **List (append)**: Adds elements one by one in a loop.
* **Set (add)**: Adds elements one by one, showing set behavior (unordered, unique values).
* **Tuple**: Attempting to append fails (commented out) since tuples are immutable.

**🧠 Key Learnings**

* **List**: Mutable, ordered, allows duplicates.
* **Tuple**: Immutable, ordered, allows duplicates.
* **Set**: Mutable, unordered, **no duplicates**.
* Use **break** and **continue** to control loop flow.
* **List comprehensions** simplify loop-based list generation.
* **Tuples can't be changed**, sets and lists can.

Let me know if you'd like a visual representation or explanation in Bangla!

**break continue**

lst = []  
for x in range(50):  
 lst.append(x)  
 print(lst)  
 if x % 2 == 0:  
 break  
  
lst = []  
for x in range(50):  
 if x % 2 == 0:  
 continue  
 lst.append(x)  
 print(lst)  
# [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49]  
  
# Important Note:

**break** like with cute girl relationship, after relation breakup occurs (after print then break)

**continue** is when girl not cute then skip and print another.

😂 Love the analogy! Here's a summary with your fun twist included:

**🔁 Code Behavior with break and continue**

**🔹 First Loop:**

python

CopyEdit

lst = []

for x in range(50):

lst.append(x)

print(lst)

if x % 2 == 0:

break

* Starts from 0.
* Appends x to the list and prints.
* **Breaks immediately on first even number (0)** — so it stops after the first iteration.

📝 **Analogy**:

break is like a **cute girl relationship** — everything is going great, but then 💔 **breakup** happens right after one interaction (print and done).

**🔹 Second Loop:**

python

CopyEdit

lst = []

for x in range(50):

if x % 2 == 0:

continue

lst.append(x)

print(lst)

* Skips even numbers.
* Appends **only odd numbers** to the list and prints.

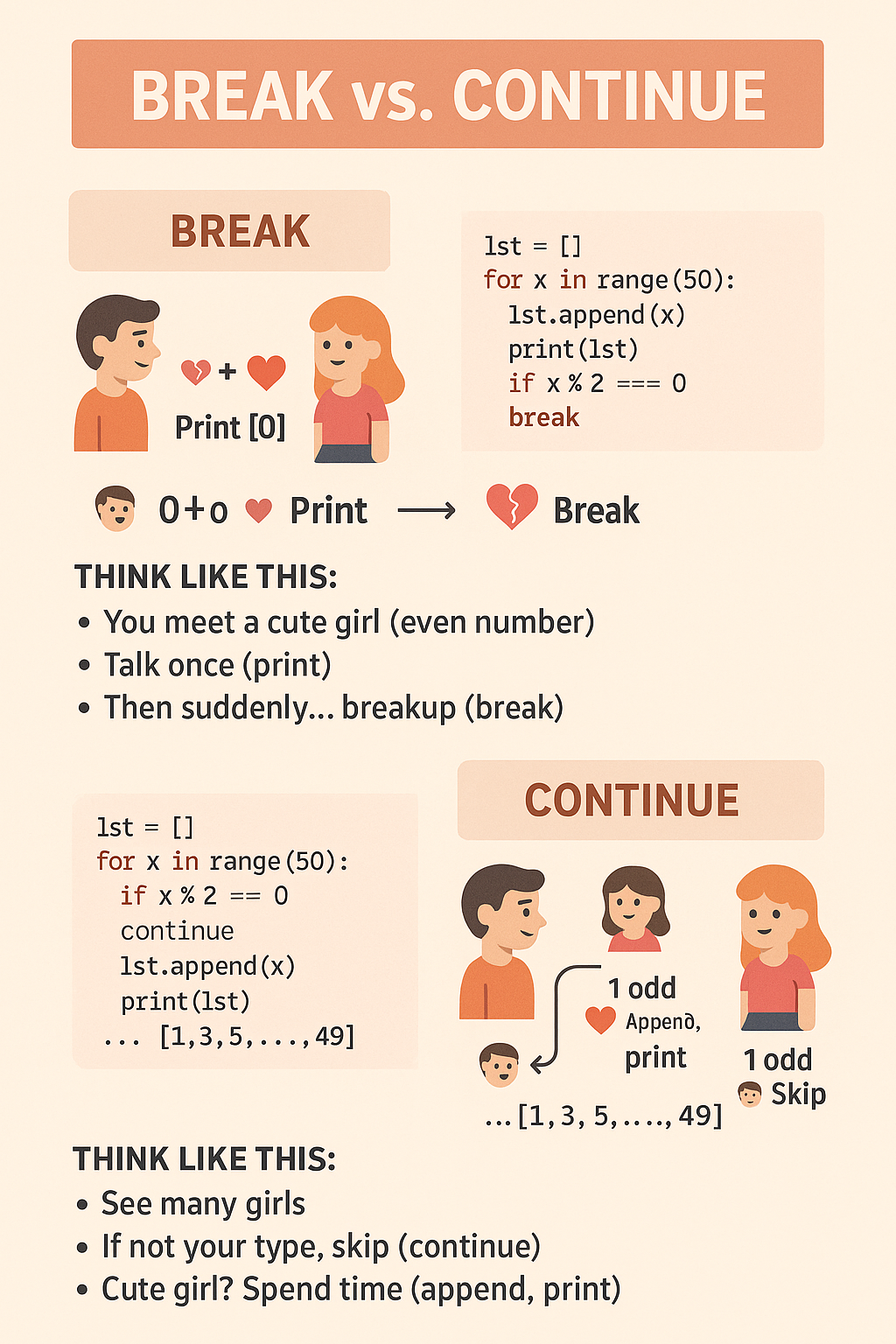
📝 **Analogy**:

continue is like seeing a **not-so-cute girl** — you politely skip 😅 and move on to the next until you find someone (an odd number) to hang out with.

**😄 Final Output:**

* First loop: [0] and done.
* Second loop: gradually prints list with odd numbers up to 49.

Let me know if you want these concepts explained with emoji graphics or real-life scenarios too!



**# list**  
lst = [x for x in range(10)]  
print(lst)  
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
lst = [3 \* x + 2 for x in range(10)]  
print(lst)  
# [2, 5, 8, 11, 14, 17, 20, 23, 26, 29]  
  
lst = [x for x in range(10)]  
print(lst[0:5])  
# [0, 1, 2, 3, 4]  
print(lst[10:0:-1])  
# [9, 8, 7, 6, 5, 4, 3, 2, 1]  
**print(lst[8:4:-1])**  
# [8, 7, 6, 5]  
  
# Second Slice: lst[10:0:-1]  
**# Syntax: lst[start: stop: step] (step = -1, meaning reverse traversal)**  
# Behavior:  
# Start (10): Out of bounds (max valid index is 9).  
# Python adjusts it to the last index (9).  
# Stop (0): The slice stops before reaching index 0 (exclusive).  
# Step (-1): Moves backward (right to left).  
# Indices included: 9, 8, 7, 6, 5, 4, 3, 2, 1 → Values: [9, 8, 7, 6, 5, 4, 3, 2, 1]  
  
lst = []  
for x in range(50):  
 lst.append(x)  
 print(lst)  
 if x % 2 == 0:  
  **break**  
  
lst = []  
for x in range(50):  
 if x % 2 == 0:  
  **continue**  
 lst.append(x)  
 print(lst)  
# [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49]  
  
**# Important Note: break like with cute girl relationship, after relation breakup occurs:  
# (after print then break) , continue is when girl not cute then skip and print another.**  
**lst = []**  
x = 20  
i = 10  
while i <= x:  
 lst.append(i)  
 i += 1  
 print(lst)  
# [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]  
 **lst1 = [x for x in range(10) if x % 2 == 0]**  
lst2 = ['mahade', 'hasan', 'jony']  
lst1.append(lst2)  
print(lst1)  
# [0, 2, 4, 6, 8, ['mahade', 'hasan', 'jony']]  
  
**# membership testing**  
lst = [x for x in range(1000)]  
print(10001 in lst)  
# False  
  
**# list insert()**  
lst = [10, 20, 30, 40, 50, 'mahade', 'hasan', 'Jony']  
lst.insert(1, 500)  
print(lst)  
# [10, 500, 20, 30, 40, 50, 'mahade', 'hasan', 'Jony']  
  
**# list append()**  
lst = [10, 20, 30, 40, 50, 'mahade']  
lst.append('hasan')  
print(lst)  
# [10, 20, 30, 40, 50, 'mahade', 'hasan']  
lst2 = ['bonna', 'badhan']  
lst.append(lst2)  
print(lst)  
# [10, 20, 30, 40, 50, 'mahade', 'hasan', ['bonna', 'badhan']]  
**print(len(lst))**  
# 8  
**# Your code uses append(), which adds the entire lst2 as a single nested list, making the length   
# If you wanted to merge the lists, you should use extend(), which would make the length 9.  
lst.extend(lst2)**  
print(lst)  
# [10, 20, 30, 40, 50, 'mahade', 'hasan', ['bonna', 'badhan'], 'bonna', 'badhan']  
print(len(lst))  
# 10  
  
**# list remove**  
lst = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]  
lst.remove(20)  
print(lst)  
# [10, 30, 40, 50, 60, 70, 80, 90, 100]  
**# list pop**  
lst.pop()  
print(lst)  
# [10, 30, 40, 50, 60, 70, 80, 90]  
**# list del**  
del lst[2]  
print(lst)  
# [10, 30, 50, 60, 70, 80, 90]  
  
lst = [x for x in range(10)]  
print(lst)  
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
**# list reverse**  
lst.reverse()  
print(lst)  
# [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]  
# list count  
**print(lst.count(1))**  
# 1  
  
lstX = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 'Copy']  
**lstY = lstX.copy()**  
print(lstY)  
# [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 'Copy']  
**lstY.reverse()**  
print(lstY)  
# ['Copy', 100, 90, 80, 70, 60, 50, 40, 30, 20, 10]  
  
lst = [10, 30, 20, 80, 70, 90, 100, 110, 50, 40, 60, 30]  
**lst.sort()**  
print(lst)  
# [10, 20, 30, 30, 40, 50, 60, 70, 80, 90, 100, 110]  
 **lst.sort(reverse=True)**  
print(lst)  
# [110, 100, 90, 80, 70, 60, 50, 40, 30, 30, 20, 10]  
  
# immutable - mutable process  
tpl = (10, 20, 30, 'mahade', 'hasan', 100)  
**lstTemp = list(tpl)**  
lstTemp.append(10)  
print(lstTemp)  
# [10, 20, 30, 'mahade', 'hasan', 100, 10]  
lstTemp.extend([20])  
print(lstTemp)  
# [10, 20, 30, 'mahade', 'hasan', 100, 10, 20]  
**# The extend() method expects an iterable (like a list, tuple, string), but you're passing an integer (20), which is not iterable. Hence, Python throws a TypeError.**  
tpl = tuple(lstTemp)  
print(tpl)  
# (10, 20, 30, 'mahade', 'hasan', 100, 10, 20)  
print(type(tpl))  
# <class 'tuple'>  
print(type(lstTemp))  
# <class 'list'>  
print(tpl[2])  
# 30  
tpl = (10, 20, 30, [100, 110], 'mahade', 'hasan')  
**print(len(tpl))**  
# 6  
print(tpl[0:3])  
# (10, 20, 30)  
print(tpl[0:4])  
print(tpl[6:0:-1])  
# ('hasan', 'mahade', [100, 110], 30, 20)  
  
**# mutable-immutable**  
lstnew = [10, 20, 30, 'mahade', 'hasan']  
print(lstnew)  
print(type(lstnew))  
lstnew.append('afra')  
tplnew = tuple(lstnew)  
print(tplnew)  
print(type(tplnew))  
  
# [10, 20, 30, 'mahade', 'hasan']  
# <class 'list'>  
# (10, 20, 30, 'mahade', 'hasan', 'afra')  
# <class 'tuple'>  
  
tpl1 = (10, 20, 30, 40, 50)  
tpl2 = (60, 70, 80, 90, 100)  
  
tpl3 = tpl1 + tpl2  
print(tpl3)  
# (10, 20, 30, 40, 50, 60, 70, 80, 90, 100)  
 **# membership check**  
print(60 in tpl3)  
# True  
del tpl2  
print(tpl3)  
  
tpl1 = (10, 20, 30, 40, 50)  
tpl2 = (60, 70, 80, 90, 100)  
  
**# tpl3 = tpl1 + tpl2 # This creates a new tuple and assigns it to tpl3  
# del tpl2 # This deletes the variable tpl2, not its contents from tpl3  
# print(tpl3) # This is perfectly valid and will print the combined tuple**

You've shared a comprehensive set of Python list and tuple operations — perfect for both beginners and intermediates. Here's a summarized **analysis and key takeaways** based on your code:

**✅ List Comprehension**

lst = [x for x in range(10)]

# [0, 1, 2, ..., 9]

* Creates lists efficiently using a single line.
* Can include conditions or expressions like [3 \* x + 2 for x in range(10)].

**✅ List Slicing and Reversing**

lst[0:5] # first 5 elements

lst[10:0:-1] # reversed from index 9 to 1

lst[8:4:-1] # reversed from index 8 to 5

* Python slicing: lst[start: stop: step] → excludes the stop index.
* Negative steps reverse the list.

**✅ Loop Control: break vs continue**

if x % 2 == 0:

break # exit loop

if x % 2 == 0:

continue # skip iteration

* break: exits the loop immediately (like a breakup 😅).
* continue: skips current iteration (like ignoring someone).

**✅ While Loop Example**

i = 10

while i <= 20:

lst.append(i)

* Adds values from 10 to 20.

**✅ Appending Nested Lists**

lst1.append(lst2)

# results in a nested list

* Use append() for adding a single element (even a list).
* Use extend() to merge two lists (flattened).

**✅ Membership Testing**

print(10001 in lst) # False

* Efficient for checking presence of elements.

**✅ List Methods Recap**

* insert(index, value): Adds at a specific index.
* append(value): Adds a value at the end.
* extend(list): Merges another iterable into the list.
* remove(value): Deletes first occurrence.
* pop(): Removes and returns the last item.
* del lst[index]: Deletes item by index.
* reverse(): Reverses the list.
* count(value): Counts occurrences.
* sort() / sort(reverse=True): Sorts in ascending/descending order.
* copy(): Creates a shallow copy.

**✅ Tuple to List Conversion and Back**

lstTemp = list(tpl)

lstTemp.append(10)

tpl = tuple(lstTemp)

* Use list() to convert a tuple for modification.
* Convert back to tuple() when immutability is needed again.

**✅ Tuple Operations**

* Tuples are immutable.
* You can concatenate (+) tuples.
* Deleting the variable (del tpl2) doesn’t affect other tuples.

**🔥 Funny Analogy:**

"break is like breaking up with a cute girl – it's over.  
continue is like skipping a girl you don't like – move on to the next."

Love that 😄 – makes remembering easier!

**🧠 Key Takeaways:**

* Use **list comprehensions** for elegant list generation.
* Understand **slicing rules** (especially with negative steps).
* Know the difference between **append vs extend**.
* Understand **mutable (list)** vs **immutable (tuple)**.
* Python’s **list methods** offer powerful tools for data manipulation.
* Use **conversion between list and tuple** when needed to modify immutable structures.

# set dictionary

# A set is an unordered, mutable, and unique collection of unique elements in Python.

# Key Properties:

# No duplicates: {1, 2, 2, 3} → {1, 2, 3}

# Unordered: Items have no fixed position.

# Mutable: You can add/remove items, but elements must be immutable (like numbers, strings, tuples).

a = {1, 2, 3}

b = {3, 4, 5}

**print(a | b) # Union: {1, 2, 3, 4, 5}**

**print(a & b) # Intersection: {3}**

**print(a - b) # Difference: {1, 2}**

**print(a ^ b) # Symmetric Difference: {1, 2, 4, 5}**

a = {10, 20, 30, 40, 50}

b = {50, 60, 70, 80, 90, 100}

c = a | b

print(c)

# {100, 70, 40, 10, 80, 50, 20, 90, 60, 30}

print(sorted(c))

# [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# Sets do not guarantee order — it's by design (for performance).

# Use sorted(set) if you want ordered output.

c = a & b

print(c)

# {50}

c = a - b

print(c)

# {40, 10, 20, 30}

c = a ^ b

print(c)

print(sorted(c))

# [10, 20, 30, 40, 60, 70, 80, 90, 100]

**# st = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, {'mahade', 'hasan', 'jony'}}**

**# print(st)**

**# Here, you're trying to add a set inside another set.**

**# But sets are unhashable, and only hashable (immutable) objects can go inside a set.**

**# ✅ How to Fix It:**

**# You can convert the inner set to a tuple, which is hashable:**

st = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, ('mahade', 'hasan', 'jony'), True, False}

print(st)

# {False, True, 100, 70, 40, 10, ('mahade', 'hasan', 'jony'), 80, 50, 20, 90, 60, 30}

# Now it works because a tuple is immutable and hashable.

st.add(500)

print(st)

# {False, True, 100, 90, 70, 40, 10, 80, 50, 20, 500, ('mahade', 'hasan', 'jony'), 60, 30}

st.remove(20)

print(st)

print(20 in st)

# {False, True, 100, 70, 40, 10, 80, 50, 500, ('mahade', 'hasan', 'jony'), 90, 60, 30}

# False

st.clear()

print(st)

# set()

**# frozenset**

**# What is a frozenset?**

**# A frozenset is just like a regular set, but it is immutable — meaning:**

# ❌ You cannot add, remove, or update its elements.

# ✅ You can still perform set operations like union, intersection, etc.

norm\_set = {10, 20, 30, 40, 50, 'mahade', 'hasan'}

frozen = frozenset(norm\_set)

print(frozen)

print(type(frozen))

# frozenset({'mahade', 50, 20, 'hasan', 40, 10, 30})

# <class 'frozenset'>

# dictionary

dct = {'1': 'mahade', '2': 'hasan', '3': 'Jony'}

print(dct)

# {'1': 'mahade', '2': 'hasan', '3': 'Jony'}

print(dct.keys())

# dict\_keys(['1', '2', '3'])

print(dct.values())

# dict\_values(['mahade', 'hasan', 'Jony'])

dct2 = dct.copy()

print(dct2)

# {'1': 'mahade', '2': 'hasan', '3': 'Jony'}

# value change

dct2['3'] = 'afra'

print(dct2)

# {'1': 'mahade', '2': 'hasan', '3': 'afra'}

dct2['4'] = 'badhan'

print(dct2)

# {'1': 'mahade', '2': 'hasan', '3': 'afra', '4': 'badhan'}

del dct2['1']

print(dct2)

# {'2': 'hasan', '3': 'afra', '4': 'badhan'}

dct2.popitem()

print(dct2)

# {'2': 'hasan', '3': 'afra'}

Here's a **short and clear summary** of the key concepts:

**🔹 Set in Python**

* **Unordered**, **mutable**, and **only unique** elements.
* Elements must be **hashable** (immutable like int, str, tuple).
* Fast lookup & operations using hash tables.

**✅ Common Set Operations:**

a = {1, 2, 3}

b = {3, 4, 5}

a | b # Union → {1, 2, 3, 4, 5}

a & b # Intersection → {3}

a - b # Difference → {1, 2}

a ^ b # Symmetric Difference → {1, 2, 4, 5}

**⚠️ Note:**

* Sets are **unordered** → use sorted(set) for ordered view.
* Cannot add mutable types like lists/sets.

**🧊 Frozenset**

* Immutable version of a set.
* Can be used as a **key in a dict** or inside a set.

fs = frozenset([1, 2, 3])

**📚 Dictionary (dict)**

* Stores data in **key-value** pairs.
* Keys must be unique and hashable.
* Since Python 3.7, maintains insertion order.

**✅ Common Methods:**

d['key'] # Access

d.get('key') # Safe access

d.keys(), d.values(), d.items()

d.pop('key'), d.popitem()

**🧠 Quick Comparison:**

| **Feature** | **Set** | **Frozenset** | **Dict** |
| --- | --- | --- | --- |
| Mutable | Yes | No | Yes |
| Ordered | No | No | Yes (3.7+) |
| Duplicates | No | No | Keys only |
| Hashable | No | Yes | Keys only |

**Array in Python – Quick Overview**

In Python, you usually use **lists** as dynamic arrays. But for performance or type-restricted arrays, you can use the **array module** or libraries like **NumPy**.

**🔹 1. List (Most Common)**

arr = [10, 20, 30, 40]

arr.append(50)

arr.remove(20)

print(arr) # [10, 30, 40, 50]

* Can hold **mixed data types**.
* Flexible and powerful.
* Built-in and widely used.

**🔹 2. array Module (from array module)**

import array

arr = array.array('i', [10, 20, 30]) # 'i' → integer

arr.append(40)

print(arr) # array('i', [10, 20, 30, 40])

* **Type-restricted** (only one data type).
* More **memory-efficient** than list for large numeric data.
* Needs to import array module.
* Common type codes: 'i' (int), 'f' (float), 'u' (Unicode char), etc.

**🔹 3. NumPy Array (for large data/scientific computing)**

import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr \* 2) # [2 4 6 8]

* **Faster** and **multi-dimensional** support.
* Rich operations: matrix, broadcasting, stats, etc.
* Requires installing NumPy (pip install numpy).

**🧠 Summary Table:**

| **Feature** | **List** | **array.array** | **NumPy array** |
| --- | --- | --- | --- |
| Built-in | ✅ | ❌ (needs import) | ❌ (needs NumPy) |
| Type restriction | ❌ | ✅ | ✅ |
| Speed | Moderate | Fast | Very Fast |
| Multi-dimension | ❌ | ❌ | ✅ |
| Best for | General use | Numeric data | Data science, ML |