# Data Structures

## **Functions**

Fibonacci Series

***fib*** *function prints the Fibonacci series up to a given limit* ***n****.*

*Uses a while loop to generate and print the series.*

**def fib(n):**

**"""Print a Fibonacci series up to n."""**

**a, b = 0, 1**

**while a < n:**

**print(a, end=' ')**

**a, b = b, a + b**

**print()**

**fib(2000)**

Function calls

**fib** # Output: <function fib at 10042ed0>

**f = fib**

**f(100)**

**fib(0)** # Output: None

**print(fib(0))** # Output: None

Function Returning List

***fib2*** *returns a list containing the Fibonacci series up to a given limit* ***n****.*

**def fib2(n):**

**"""Return a list containing the Fibonacci series up to n."""**

**result = []**

**a, b = 0, 1**

**while a < n:**

**result.append(a)**

**a, b = b, a + b**

**return result**

**f100 = fib2(100)**

**f100**  # Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

Return Statement and List Method append()

**return** statement returns a value; without an expression, it's **None**.

**result.append(a)** adds an element to the list **result**.

More efficient than using **result = result + [a].**

## **List Basics**

Creating List

# empty list

**my\_list = []**

# list of integers

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

# list with mixed data types

**my\_list = [1, "Hello", 3.4]**

# nested list

**my\_list = ["mouse", [8, 4, 6], ['a']]**

Accessing List Elements

Use the index operator **[]** to access items in a list. Indices start at 0 in Python.

Accessing an index beyond the list's range raises an **IndexError**.

Indices must be integers; using other types results in a **TypeError**.

Nested lists are accessed using nested indexing.

**my\_list = ['p', 'r', 'o', 'b', 'e']**

# Output: p

**print(my\_list[0])**

# Output: o

**print(my\_list[2])**

# Output: e

**print(my\_list[4])**

# Nested List

**n\_list = ["Happy", [2, 0, 1, 5]]**

# Nested indexing

**print(n\_list[0][1])**

**print(n\_list[1][3])**

# Error! Only integers can be used for indexing

**print(my\_list[4.0])**

# Output

**p**

**o**

**e**

**a**

**5**

Traceback (most recent call last):

File "<string>", line 21, in <module>

TypeError: list indices must be integers or slices, not float

Attempting to use a non-integer index, like a float, results in a **TypeError**

Negative Indexing

# Negative indexing in lists

**my\_list = ['p','r','o','b','e']**

**print(my\_list[-1])**

**print(my\_list[-5])**

# Output: e, p

## **Slice Lists in Python**

We can access a range of items in a list by using the slicing operator :(colon).

# List slicing in Python

my\_list = ['p','r','o','g','r','a','m','i','z']

# elements 3rd to 5th

print(my\_list[2:5])

# elements beginning to 4th

print(my\_list[:-5])

# elements 6th to end

print(my\_list[5:])

# elements beginning to end

print(my\_list[:])

# Output

['o', 'g', 'r']

['p', 'r', 'o', 'g']

['a', 'm', 'i', 'z']

['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']

## **Lists Methods**

List Methods

**list.append(x)**

Adds an item to the end of the list.

Equivalent to **a[len(a):] = [x]**

**list.extend(iterable)**

Extends the list by appending all items from the iterable.

Equivalent to **a[len(a):] = iterable**

**list.insert(i, x)**

Inserts an item at a given position.

The first argument is the index of the element before which to insert.

**list.remove(x)**

Removes the first item from the list whose value is equal to x.

Raises a ValueError if there is no such item.

**list.pop([i])**

Removes and returns the item at the given position in the list.

If no index is specified, removes and returns the last item.

**list.clear()**

Removes all items from the list.

Equivalent to **del a[:]**

**list.index(x[, start[, end]])**

Returns the zero-based index in the list of the first item whose value is equal to x.

Raises a ValueError if there is no such item.

Optional arguments **start** and **end** limit the search to a particular subsequence.

**list.count(x)**

Returns the number of times x appears in the list

**list.sort(key=None, reverse=False)**

Sorts the items of the list in place.

Optional arguments for sort customization.

**list.reverse()**

Reverses the elements of the list in place.

**list.copy()**

Returns a shallow copy of the list.

Equivalent to **a[:]**

Example

**fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']**

**fruits.count('apple')** # Output: 2

**fruits.count('tangerine')** # Output: 0

**fruits.index('banana')** # Output: 3

**fruits.index('banana', 4)** # Find next banana starting at position 4, Output: 6

**fruits.reverse()**

# Output: ['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']

**fruits.append('grape')**

# Output: ['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']

**fruits.sort()**

# Output: ['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']

**fruits.pop()** # Output: 'pear'

Change/Add List elements

Lists are mutable, meaning their elements can be changed unlike [string](https://www.programiz.com/python-programming/string) or [tuple](https://www.programiz.com/python-programming/tuple).

We can use the assignment operator = to change an item or a range of items.

# Correcting mistake values in a list

**odd = [2, 4, 6, 8]**

# change the 1st item

**odd[0] = 1**

**print(odd)**

# change 2nd to 4th items

**odd[1:4] = [3, 5, 7]**

**print(odd)**

# Output

[1, 4, 6, 8]

[1, 3, 5, 7]

# Appending and Extending lists in Python

**odd = [1, 3, 5]**

**odd.append(7)**

**print(odd)**

**odd.extend([9, 11, 13])**

**print(odd)**

# Output

[1, 3, 5, 7]

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

# Concatenating and repeating lists

**odd = [1, 3, 5]**

**print(odd + [9, 7, 5])**

**print(["re"] \* 3)**

# Output

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

['re', 're', 're']

# list insert() method

**odd = [1, 9]**

**odd.insert(1,3)**

**print(odd)**

**odd[2:2] = [5, 7]**

**print(odd)**

# Output

[1, 3, 9]

[1, 3, 5, 7, 9]

Delete/Remove List Items

# Deleting list items

**my\_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']**

# delete one item

**del my\_list[2]**

**print(my\_list)**

# delete multiple items

**del my\_list[1:5]**

**print(my\_list)**

# delete entire list

**del my\_list**

# Error: List not defined

**print(my\_list)**

**# Output**

**['p', 'r', 'b', 'l', 'e', 'm']**

**['p', 'm']**

**Traceback (most recent call last):**

**File "<string>", line 18, in <module>**

**NameError: name 'my\_list' is not defined**

Clear method

**my\_list = ['p','r','o','b','l','e','m']**

**my\_list.remove('p')**

**# Output: ['r', 'o', 'b', 'l', 'e', 'm']**

**print(my\_list)**

**# Output: 'o'**

**print(my\_list.pop(1))**

**# Output: ['r', 'b', 'l', 'e', 'm']**

**print(my\_list)**

**# Output: 'm'**

**print(my\_list.pop())**

**# Output: ['r', 'b', 'l', 'e']**

**print(my\_list)**

**my\_list.clear()**

**# Output: []**

**print(my\_list)**

Deleting items in list by assigning empty list to a slice of elements

**my\_list = ['p','r','o','b','l','e','m']**

**my\_list[2:3] = []**

**my\_list**

**['p', 'r', 'b', 'l', 'e', 'm']**

**my\_list[2:5] = []**

**my\_list**

**['p', 'r', 'm']**

## **Using Lists as Stack (Last-In, First-Out)**

Lists can be used as a stack using **append()** to add an item to the top and **pop()** without an index to retrieve an item from the top.

The stack follows the last-in, first-out (LIFO) principle

**stack = [3, 4, 5]**

**stack.append(6)**

**stack.append(7)**

# Output: [3, 4, 5, 6, 7]

**stack.pop() # Output: 7**

# Output: [3, 4, 5, 6]

**stack.pop()** # Output: 6

**stack.pop()** # Output: 5

# Output: [3, 4]

## **Using Lists as Queue (First-In, First-Out)**

While lists can be used as a queue, **collections.deque** is more efficient for this purpose.

**deque** allows fast appends and pops from both ends.

The queue follows the first-in, first-out (FIFO) principle.

**from collections import deque**

**queue = deque(["Eric", "John", "Michael"])**

**queue.append("Terry")** # Terry arrives

**queue.append("Graham")** # Graham arrives

**queue.popleft()** # The first to arrive now leaves

# Output: 'Eric'

**queue.popleft()** # The second to arrive now leaves

# Output: 'John'

# Remaining queue in order of arrival

# Output: deque(['Michael', 'Terry', 'Graham'])

## **List Comprehensions**

List comprehensions provide a concise way to create lists. Common applications include creating new lists or subsequences based on operations applied to each member of a sequence or iterable.

A list comprehension consists of brackets containing an expression followed by a **for** clause.

Optionally, it can have zero or more **for** or **if** clauses.

Square of numbers

# Using a loop

**squares = []**

**for x in range(10):**

**squares.append(x\*\*2)**

# Using list comprehension

**squares = [x\*\*2 for x in range(10)]**

# Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Combining Elements from Two Lists

# Using nested loops

**combs = []**

**for x in [1, 2, 3]:**

**for y in [3, 1, 4]:**

**if x != y:**

**combs.append((x, y))**

# Using list comprehension with condition

**combs=[(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]**

# Output: [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

Applying Expressions to Elements

vec = [-4, -2, 0, 2, 4]

# Using list comprehension with expressions

**[x\*2 for x in vec]** # Output: [-8, -4, 0, 4, 8]

**[abs(x) for x in vec]** # Output: [4, 2, 0, 2, 4]

Calling Methods on Elements

**freshfruit=[' banana', ' loganberry ', 'passion fruit ']**

# Using list comprehension with methods

**[weapon.strip() for weapon in freshfruit]**

# Output: ['banana', 'loganberry', 'passion fruit']

Nested List Comprehension

**vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]**

# Using nested list comprehension

**[num for elem in vec for num in elem]**

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

Complex Expressions and Nested Functions

**from math import pi**

# Using list comprehension with complex expressions

**[str(round(pi, i)) for i in range(1, 6)]**

# Output: ['3.1', '3.14', '3.142', '3.1416', '3.14159']

## **List Comprehension with Nested Expressions**

Transposing Rows and Columns of a Matrix

# Matrix: 3x4 matrix implemented as a list of 3 lists of length 4

**matrix = [**

**[1, 2, 3, 4],**

**[5, 6, 7, 8],**

**[9, 10, 11, 12],**

**]**

# Using list comprehension to transpose rows and columns

**[[row[i] for row in matrix] for i in range(4)]**

# Output: [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

Equivalent Loop Implementation

**transposed = []**

**for i in range(4):**

**transposed.append([row[i] for row in matrix])**

# Output: [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

Equivalent Loop Implementation

**transposed = []**

**for i in range(4):**

**transposed.append([row[i] for row in matrix])**

# Output: [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

Equivalent Nested Loop Implementation

**transposed = []**

**for i in range(4):**

**transposed\_row = []**

**for row in matrix:**

**transposed\_row.append(row[i])**

**transposed.append(transposed\_row)**

# Output: [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

Using zip() Function

# Using zip() function to transpose rows and columns

**list(zip(\*matrix))**

# Output: [(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]

## **The del statement / Tuples and Sequences**

**del** statement removes items from a list based on index, differing from **pop()** which returns a value.

It can remove slices from a list or clear the entire list.

Removing Items from a List with del

# List

**a = [-1, 1, 66.25, 333, 333, 1234.5]**

# Using del to remove items by index

**del a[0]** # Removes the first element

# Output: [1, 66.25, 333, 333, 1234.5]

**del a[2:4]** # Removes elements at index 2 and 3

# Output: [1, 66.25, 1234.5]

**del a[:]** **/ del a** # Clears the entire list

# Output: []

Tuples

Tuples and lists have similarities but are used in different situations.

Tuples are often used for heterogeneous sequences, accessed via unpacking or indexing.

Lists are mutable and typically contain homogeneous elements, accessed by iteration.

Tuples are constructed using parentheses **( )**.

They are immutable, meaning their elements cannot be changed after creation.

Tuples may contain mutable objects like lists.

# Tuple

**t = 12345, 54321, 'hello!'**

# Accessing elements

**t[0]** # Output: 12345

# Output: (12345, 54321, 'hello!')

# Nested tuple

**u = t, (1, 2, 3, 4, 5)**

# Output: ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))

# Tuples are immutable

**t[0] = 88888** # Results in a TypeError

# Tuples can contain mutable objects

**v = ([1, 2, 3], [3, 2, 1])**

# Output: ([1, 2, 3], [3, 2, 1])

Creating a Tuple

**empty = ()** # Empty tuple

**singleton = 'hello',** # Tuple with one item

# Output: ('hello',)

# Pack a tuple and assign to variable

**x, y, z = t**

## **Slice Tuple**

A slice in Python refers to a subset of elements from a sequence, such as a list, tuple, or string.

Slicing allows you to extract a portion of the sequence by specifying a start index, an end index, and an optional step size.

The basic syntax for slicing is as follows: **sequence[start:stop:step]**

**start**: The index from which the slice begins (inclusive)

**stop**: The index at which the slice ends (exclusive)

**step** (optional): The step size, indicating the interval between elements. Default is 1.

Slice examples

**my\_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**

# Basic slices

**subset = my\_list[2:6]** # Elements at index 2, 3, 4, 5

# Output: [2, 3, 4, 5]

# Slicing with step

**subset\_step = my\_list[1:8:2]** # Elements at index 1, 3, 5, 7

# Output: [1, 3, 5, 7]

# Omitting start and stop

**subset\_all = my\_list[:]** # Copy of the entire list

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

# Slicing strings

**text = "Hello, World!"**

**substring = text[7:12]** # Characters at index 7, 8, 9, 10, 11

# Output: "World"

## **Sets**

A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries.

Set objects support mathematical operations like union, intersection, difference, and symmetric difference.

Use curly braces **{}** or the **set()** function to create sets. To create an empty set, use **set()**, not **{}**, as the latter creates an empty dictionary.

# Creating a set

**basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}**

# Output: {'orange', 'banana', 'pear', 'apple'}

# Set operations

**a = set('abracadabra')**

**b = set('alacazam')**

# Output: {'a', 'c'}

**a - b** # letters in a but not in b

# Output: {'a', 'c'}

**a & b** # letters in both a and b

# Set comprehensions

**a = {x for x in 'abracadabra' if x not in 'abc'}**

# Output: {'r', 'd'}

## **Dictionary**

A dictionary is a collection of key-value pairs, where keys are unique and can be of any immutable type.

Key-value pairs are enclosed in braces **{}** and separated by commas.

The main operations include storing a value with a key, extracting a value given a key, and deleting a key-value pair.

Dictionaries support the **in** keyword for checking key existence.

The **dict()** constructor and dictionary comprehensions are ways to create dictionaries.

# Creating a dictionary

**tel = {'jack': 4098, 'sape': 4139}**

# Adding, accessing, and deleting key-value pairs

**tel['guido'] = 4127**

**tel['jack']** # Output: 4098

**del tel['sape']**

# Output: {'jack': 4098, 'guido': 4127}

# Dict constructor and comprehension

**dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])**

# Output: {'sape': 4139, 'guido': 4127, 'jack': 4098}

**{x: x\*\*2 for x in (2, 4, 6)}**

# Output: {2: 4, 4: 16, 6: 36}

## **Looping Techniques**

Iterating Over Dictionaries

**knights = {'gallahad': 'the pure', 'robin': 'the brave'}**

**for k, v in knights.items():**

**print(k, v)**

# Output

gallahad the pure

robin the brave

Enumerating Sequences

Use the **enumerate()** function to get both the position index and corresponding value.

**for i, v in enumerate(['tic', 'tac', 'toe']):**

**print(i, v)**

# Output

0 tic

1 tac

2 toe

Looping Over Multiple Sequences

Pair entries from different sequences using the **zip()** function.

**questions = ['name', 'quest', 'favorite color']**

**answers = ['lancelot', 'the holy grail', 'blue']**

**for q, a in zip(questions, answers):**

**print('What is your {0}? It is {1}.'.format(q, a))**

# Output

What is your name? It is lancelot.

What is your quest? It is the holy grail.

What is your favorite color? It is blue.

Reversing and Sorting Sequences

To loop in reverse, use **reversed()** on the sequence.

To loop in sorted order, use **sorted()** function.

# Reverse loop

**for i in reversed(range(1, 10, 2)):**

**print(i)**

# Output

9

7

5

3

1

# Sorted loop

**basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']**

**for f in sorted(set(basket)):**

**print(f)**

# Output

apple

banana

orange

Pear

Modifying a List During Iteration

It is often safer to create a new list while looping rather than modifying the existing one

import math

**raw\_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]**

**filtered\_data = [value for value in raw\_data if not math.isnan(value)]**

# Output

[56.2, 51.7, 55.3, 52.5, 47.8]

## **Conditions in While and If Statements**

General Comparison Operators

Conditions in **while** and **if** statements can use any operators, not just comparisons.

The operators **in** and **not in** check for the presence or absence of a value in a sequence.

The operators **is** and **is not** compare whether two objects are the same (matters for mutable objects like lists).

All comparison operators have equal priority, lower than numerical operators.

Chained Comparisons

Comparisons can be chained, e.g., **a < b == c** tests whether **a** is less than **b** and **b** equals **c**

Use Boolean operators **and** and **or** to combine comparisons. The outcome of a comparison can be negated with **not**.

Priority: **not** > **and** > **or**, but parentheses can be used for clarity

Short-Circuit Evaluation

**And** and **or** are short-circuit operators.

Evaluation stops as soon as the outcome is determined.

Example: **A and B and C** does not evaluate **C** if **A** and **B** are true.

When used as a value, the return value of a short-circuit operator is the last evaluated argument.

Assigning Result to a Variable

You can assign the result of a comparison or Boolean expression to a variable.

**string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'**

**non\_null = string1 or string2 or string3**

# Output

**'Trondheim'**

Unlike C, assignment cannot occur inside expressions in Python.

This design choice helps avoid common problems in C programs where **=** is mistakenly used instead of **==**.

## **Comparing Sequence Objects**

Lexicographical Ordering

Sequence objects (of the same type) can be compared using lexicographical ordering.

Comparison proceeds item by item until a difference is found, determining the outcome.

If two items being compared are sequences of the same type, the comparison is carried out recursively.

If all items compare equal, the sequences are considered equal.

If one sequence is an initial sub-sequence of the other, the shorter sequence is considered smaller.

Unicode Code Point for Strings

For strings, lexicographical ordering uses the Unicode code point number for individual characters.

**(1, 2, 3) < (1, 2, 4)**

**[1, 2, 3] < [1, 2, 4]**

**'ABC' < 'C' < 'Pascal' < 'Python'**

**(1, 2, 3, 4) < (1, 2, 4)**

**(1, 2) < (1, 2, -1)**

**(1, 2, 3) == (1.0, 2.0, 3.0)**

**(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)**

Comparing Different Types

Comparing objects of different types with **<** or **>** is legal if the objects have appropriate comparison methods.

Mixed numeric types are compared based on their numeric value (e.g., 0 equals 0.0).

If objects of incompatible types are compared, a **TypeError** exception is raised.

# Errors and Exceptions

## **Syntax Errors**

Syntax errors, also known **as parsing errors**, are common during the learning phase of Python. They occur when there is a mistake in the structure of the code.

**while 1 print 'Hello world'** # print is missing a colon

**File "<stdin>", line 1**

**while 1 print 'Hello world'**

**^**

**SyntaxError: invalid syntax**

Exceptions

Exceptions occur during the execution of a syntactically correct statement or expression. They are not unconditionally fatal, and handling them in Python programs is possible

**10 \* (1/0)**

**4 + spam\*3**

**'2' + 2**

Types of exceptions: **ZeroDivisionError**, **NameError**, and **TypeError**

## **Handling Exceptions**

The **try** statement attempts to execute the code in its block. If an exception occurs, it jumps to the **except** block.

**while True:**

**try:**

**x = int(input("Please enter a number: "))**

**break**

**except ValueError:**

**print("Oops! That was no valid number. Try again...")**

Multiple Except Clauses

**except (RuntimeError, TypeError, NameError):**

**pass**

Exception Hierarchy

**class B(Exception):**

**pass**

**class C(B):**

**pass**

**class D(C):**

**pass**

**for cls in [B, C, D]:**

**try:**

**raise cls()**

**except D:**

**print("D")**

**except C:**

**print("C")**

**except B:**

**print("B")**

Exception with No Name (Wildcard):

**except:**

**print("Unexpected error:", sys.exc\_info()[0])**

**raise**

Try...Except...Else:

**try:**

**f = open('myfile.txt')**

**s = f.readline()**

**i = int(s.strip())**

**except OSError as err:**

**print("OS error: {0}".format(err))**

**except ValueError:**

**print("Could not convert data to an integer.")**

**except:**

**print("Unexpected error:", sys.exc\_info()[0])**

**raise**

**else:**

**print("No exception occurred.")**

Handling Exception Arguments

**try:**

**raise Exception('spam', 'eggs')**

**except Exception as inst:**

**print(type(inst))** # the exception instance

**print(inst.args)** # arguments stored in .args

**print(inst)** # \_\_str\_\_ allows args to be printed

**x, y = inst.args** # unpack args

**print('x =', x)**

**print('y =', y)**

Exception Inside Functions

Exceptions can be handled even if they occur inside functions called in the **try** clause.

**def this\_fails():**

**x = 1/0**

**try:**

**this\_fails()**

**except ZeroDivisionError as err:**

**print('Handling run-time error:', err)**

Output: **Handling run-time error: division by zero**

## **Raising an Exception**

Raising an Exception:

**x = 10**

**if x > 5:**

**raise Exception('x should not exceed 5. The value of x was: {}'.format(x))**

AssertionError Exception:

**def KelvinToFahrenheit(Temperature):**

**assert (Temperature >= 0), "Colder than absolute zero!"**

**return ((Temperature-273)\*1.8)+32**

**print(KelvinToFahrenheit(273))**

**print(int(KelvinToFahrenheit(505.78)))**

**print(KelvinToFahrenheit(-5))**

Try and Except Block

**try:**

**fh = open("testfile", "w")**

**fh.write("This is my test file for exception handling!!")**

**except IOError:**

**print("Error: can't find file or read data")**

**else:**

**print("Written content in the file successfully")**

**fh.close()**

Written content in the file successfully

Try, Except, and Finally Block

**try:**

**fh = open("testfile", "w")**

**fh.write("This is my test file for exception handling!!")**

**finally:**

**print("Error: can't find file or read data")**

**fh.close()**

Error: can't find file or read data

User-Defined Exceptions:

class Networkerror(RuntimeError):

def \_\_init\_\_(self, arg):

self.args = arg

try:

raise Networkerror("Bad hostname")

except Networkerror as e:

print(e.args)

Defining Clean-up Actions

try:

raise KeyboardInterrupt

finally:

print('Goodbye, world!')

Goodbye, world!

KeyboardInterrupt

More Complex Cases

If an exception occurs during execution of the **try** clause, the exception may be handled by an **except** clause. In all cases, the exception is re-raised after the **finally** clause has been executed.

An exception could occur during the execution of an **except** or **else** clause. The exception is re-raised after the **finally** clause has been executed.

If the **try** statement reaches a **break**, **continue**, or **return** statement, the **finally** clause will execute just before the **break**, **continue**, or **return** statement's execution.

If a **finally** clause includes a **return** statement, the **finally** clause's **return** statement will execute before, and instead of, the **return** statement in a **try** clause.

**def bool\_return() -> bool:**

**try:**

**return True**

**finally:**

**return False**

**bool\_return()** # Output: False

**def divide(x, y):**

**try:**

**result = x / y**

**except ZeroDivisionError:**

**print("division by zero!")**

**else:**

**print("result is", result)**

**finally:**

**print("executing finally clause")**

**divide(2, 1)**

# Output:

# result is 2.0

# executing finally clause

**divide(2, 0)**

# Output:

# division by zero!

# executing finally clause

divide("2", "1")

# Output:

# executing finally clause

# TypeError: unsupported operand type(s) for /: 'str' and 'str'

# Classes

## **Basics of Python Classes**

* Creating a new class creates a new type of object, allowing instances of that type to be created.
* Class inheritance mechanism allows multiple base classes.
* A derived class can override methods of its base class.
* Methods can call the method of a base class with the same name.
* Objects can contain arbitrary amounts and kinds of data.
* Classes are created at runtime and can be modified further after creation.

Attributes and Methods

Each class instance can have attributes for maintaining its state. Class instances can have methods for modifying their state.

## **Namespaces and Python Scope**

Namespace

A namespace is a mapping from names to objects.

include built-in names, global names in a module, local names in a function, attributes of an object.

**Attributes:** In expressions like **z.real**, **real** is considered an attribute of the object **z**.

**Scope:** A scope is a textual region of a Python program where a namespace is directly accessible.

Scopes are determined statically but used dynamically during execution.

Namespace Creation and Lifetimes

**Built-in Names:** The namespace containing built-in names is created when the Python interpreter starts and is never deleted.

**Module Global Names:** Created when a module definition is read in, lasting until the interpreter quits.

**Local Names in Function:** Created when a function is called and deleted when the function returns or raises an unhandled exception.

**Global Names in Module:** Created for the module when the module definition is read in.

**Scopes and Namespaces Interaction**

**Nested Scopes:** Three nested scopes are accessible during execution:

**Innermost scope:** local names.

**Enclosing functions:** non-local and non-global names.

**Next-to-last scope:** current module's global names.

**Outermost scope:** built-in names.

**Global and Nonlocal Declarations**

**global** declaration: Indicates variables live in the global scope.

**nonlocal** declaration: Indicates variables live in an enclosing scope.

**Scope Determination**

**Textual vs. Dynamic:** Scopes are determined textually; the global scope of a function is the module's namespace. The search for names is dynamic and happens at runtime.

**Quirk:** Assignments always go to the innermost scope by default.

**Global** and **nonlocal** statements are used to indicate specific scopes for variables.

**def scope\_test():**

**def do\_local():**

**spam = "local spam"**

**def do\_nonlocal():**

**nonlocal spam**

**spam = "nonlocal spam"**

**def do\_global():**

**global spam**

**spam = "global spam"**

**spam = "test spam"**

**do\_local()**

**print("After local assignment:", spam)**

**do\_nonlocal()**

**print("After nonlocal assignment:", spam)**

**do\_global()**

**print("After global assignment:", spam)**

**scope\_test()**

**print("In global scope:", spam)**

# Output

After local assignment: test spam

After nonlocal assignment: nonlocal spam

After global assignment: nonlocal spam

In global scope: global spam

* Local assignment doesn't change **scope\_test**'s binding of **spam**.
* Nonlocal assignment changes **scope\_test**'s binding of **spam**.
* Global assignment changes the module-level binding of **spam**.
* No previous binding for **spam** before the global assignment.

## **Creating a class and object**

Simple way to write a class

**class ClassName:**

**<statement-1>**

**.**

**.**

**.**

**<statement-N>**

Class Objects

**Namespace Creation:**

* When a class definition is entered, a new namespace is created, and local assignments go into this namespace.
* Function definitions inside a class create names for the functions in this new namespace.

**Class Object Creation:**

* When the class definition is left, a class object is created, serving as a wrapper around the class's namespace.
* The original local scope is reinstated, and the class object is bound to the class name.

Attribute Syntax

**class MyClass:**

**i = 12345**

**def f(self):**

**return 'hello world'**

# Attribute references

**MyClass.i** # Returns 12345

**MyClass.f** # Returns a function object

Class instantiation

Class instantiation is done using function notation, treating the class object as a parameterless function.

The result is a new instance of the class.

**x = MyClass()** # Creates a new instance of MyClass and assigns it to the variable x

**class Complex:**

**def \_\_init\_\_(self, realpart, imagpart):**

**self.r = realpart**

**self.i = imagpart**

# Example of instantiation with \_\_init\_\_

**x = Complex(3.0, -4.5)**

**x.r, x.i** # Returns (3.0, -4.5)

Instance Objects

Instance objects support attribute references (data attributes and methods).

# Data attribute reference

**x.counter = 1**

**while x.counter < 10:**

**x.counter = x.counter \* 2**

**print(x.counter)** # Prints 16

**del x.counter**

Method Objects

**Method calls:** Methods are functions that belong to an object. Involve passing the instance object as the first argument to the method

**x.f()** # Equivalent to MyClass.f(x)

**Method Objects:** Methods can be stored as method objects and called at a later time.

**xf = x.f**

**while True:**

**print(xf()) # Continues to print 'hello world'**

**Method Implementation:**

* When a method is called, the instance object is passed as the first argument to the function.
* The method object is created by packing the instance object and the function object together.

## **Class and Instance Variables**

**Instance Variables**

* Instance variables are for data unique to each instance of a class.
* They are defined in the **\_\_init\_\_** method and are unique to each instance.
* Accessible using **self.variable\_name**.

Class Variables:

* Class variables are for attributes and methods shared by all instances of a class.
* They are defined outside of any method and are shared among all instances.
* Accessible using **ClassName.variable\_name**.

class Dog:

**kind = 'canine'** # class variable shared by all instances

**def \_\_init\_\_(self, name):**

**self.name = name** # instance variable unique to each instance

# Creating instances

**d = Dog('Fido')**

**e = Dog('Buddy')**

# Accessing variables

**d.kind** # 'canine' (shared by all dogs)

**e.kind** # 'canine' (shared by all dogs)

**d.name** # 'Fido' (unique to d)

**e.name** # 'Buddy' (unique to e)

Shared Data and Mutable Objects

Shared data can have surprising effects, especially with mutable objects like lists.

Avoid using mutable objects as class variables unless intended.

**class Dog:**

**tricks = []** # Mistaken use of a class variable

**def \_\_init\_\_(self, name):**

**self.name = name**

**def add\_trick(self, trick):**

**self.tricks.append(trick)** # Unexpectedly shared by all dogs

# Creating instances

**d = Dog('Fido')**

**e = Dog('Buddy')**

# Adding tricks

**d.add\_trick('roll over')**

**e.add\_trick('play dead')**

# Accessing tricks

**d.tricks** # ['roll over', 'play dead'] (unexpectedly shared by all dogs)

Correct Design with Instance Variables:

Design classes to use instance variables to avoid unexpected sharing.

Correct design involves using instance variables for data unique to each instance, ensuring that mutable objects like lists are not shared among all instances.

**class Dog:**

**def \_\_init\_\_(self, name):**

**self.name = name**

**self.tricks = [] # Creates a new empty list for each dog**

**def add\_trick(self, trick):**

**self.tricks.append(trick)**

# Creating instances

**d = Dog('Fido')**

**e = Dog('Buddy')**

# Adding tricks

**d.add\_trick('roll over')**

**e.add\_trick('play dead')**

# Accessing tricks

**d.tricks** # ['roll over'] (unique to d)

**e.tricks** # ['play dead'] (unique to e)

## **Class Notes**

Override and Naming Conventions

Data attributes can override method attributes with the same name.

To avoid accidental conflicts, use conventions like capitalizing method names, prefixing data attribute names with an underscore, or using verbs for methods and nouns for data attributes.

**Data Attributes and Methods**

Data attributes can be referenced by methods and ordinary users of an object.

Python doesn't enforce data hiding; it relies on convention.

The first argument of a method is conventionally named **self**, but it has no special meaning to Python.

Client Caution

Clients should use data attributes carefully to avoid messing up invariants maintained by methods.

Clients can add their own data attributes without affecting method validity, as long as naming conflicts are avoided.

Readability

No shorthand for referencing data attributes or methods within methods increases readability.

Helps in avoiding confusion between local variables and instance variables.

**Method Definitions**

* Any function object that is a class attribute defines a method for instances of that class.
* The function definition doesn't need to be textually enclosed in the class definition.
* Methods can call other methods using method attributes of the self argument.

**class Bag:**

**def \_\_init\_\_(self):**

**self.data = []**

**def add(self, x):**

**self.data.append(x)**

**def addtwice(self, x):**

**self.add(x)**

**self.add(x)**

**Global scope and methods**

* Methods may reference global names similarly to ordinary functions.
* The global scope associated with a method is the module containing its definition.
* Each value is an object and has a class (type), stored as **object.\_\_class\_\_**

class C:

# Function defined outside the class

**def f1(self, x, y):**

**return min(x, x+y)**

**f = f1** # Assigning function to class attribute

**def g(self):**

**return 'hello world'**

**h = g** # Assigning function to class attribute

# Now, f, g, and h are all attributes of class C referring to function objects.

While global data in methods is rare, legitimate uses of the global scope include functions and modules imported into the global scope, which can be used by methods. The class containing the method is typically defined in the global scope, allowing methods to reference their own class.