Object-oriented programming: Paradigm modeling concepts as objects with attributes and methods.

Classes, objects, attributes, methods, properties:

Classes: Blueprints for creating objects.

Objects: Instances of classes.

Attributes: Data associated with classes or objects.

Methods: Functions associated with classes or objects.

Properties: Accessors and mutators for class attributes.

Object and class attributes @classmethod decorator:

@classmethod: Decorator defining methods operating on the class itself.

Class attributes shared among class instances.

Decorators and multiple constructors: "__ init __" → used to initialize an object, this is a constructor method.

Decorators modify behavior of functions. They don't modify the original function

Multiple constructors implemented using class or static methods. They set the initial state of the object being created

PEP 8:

Style guide for Python code to enhance readability. Indentation ↗4, Max Line leng ↗79, Imports, Whitespace, Blank lines ↳2 lines, Comments, Naming convention, arguments.(self / cls)

Software engineering principles:

Separation of concerns, High cohesion ↳ interconnectedness & low coupling → dependency.

Random numbers:

Module random for generating random numbers.

random.random(), random.randrange(), random.randint(), random.seed().

Slicing and comprehensions of lists, tuples, dictionaries:

Slicing: Extracting parts of sequences. [::-1] returns the reverse of the string

Comprehensions: Concise syntax for creating data structures. Better than loops.

Pythonic Programming:

Style of code that adheres to the best practices of python.

Makes code more readable, clear and efficient.

Packages, modules, files, functions, doc-strings & dunder vars:

Packages: Hierarchical file directory containing modules.

Modules: Python files containing code.

Functions: Blocks of reusable code.

Doc-strings: Documentation strings for modules, functions, classes, methods.

Dunder vars: Special variables and methods preceded and followed by double __

Collections:

Data structures like namedtuples, deque, tuple, set, list, dictionary.

Each collection has its own properties and use cases.

Types & type hints:

Data types: int, float, str, bool, complex, None, Object.

Type hints indicate expected types in functions and variables.

OOD:

Encapsulation: hiding the internals of the code from the outside world.

Abstraction: Breaking down of complex systems into smaller more manageable parts. ↑development + maintonence

Inheritance: Allows properties to be inherited from parent classes, reduces duplication.

Polymorphism: Objects taking different forms based on what's required, makes code flexible. ↑readability

Modularity: organization of code for reusability across various projects. ↑readability + maintenance

Struct types, typedefs, pointers:

Struct types: User-defined data types. used to group variables of different types

Typedefs: Alias for existing data types.

Pointers: Variables storing memory addresses.

Singly linked lists: Data structures with elements referencing the next element. Can be traversed in 1 direction

Doubly linked lists: Singly linked lists with references to next and previous elements. Can be traversed in both directions

Binary trees: Hierarchical data structures with at most two children per node. [inorder, pre-order, post-order] traversal

Dynamic data structures: Structures that can grow or shrink during execution. From the heap

Dynamic storage allocation: Allocating memory during program execution with malloc(), realloc(), calloc(), free().

Self-referential data structures: Structures containing references to themselves. helps forming complex data structures

What is Quantum Computing?

Computing paradigm utilizing quantum-mechanical phenomena.

Quantum Advantage vs Quantum Utility: outperforming normal computers

Quantum Advantage: Exponential speedup for certain problems. Qubits, Superposition, Entanglement, interference

Quantum Utility: Practical usefulness considering factors like error rates and scalability. stepping stone to Q. advantage.

Call-by-reference: Can be implemented using any data structure that is mutable input + output

Call-by-value: Values that are passed cannot be changed by the function, used by immutable objects. •input only

Mutable: State of the object can be changed after creation.

Immutable: State of the object cannot be changed after its creation.

Docstring usage: " " " docstring text " " "

Python Namespaces: LEGB – local, enclosing, global, Built-in.

IS-a → Inheritance

Part-of → Aggregation

Has-a → Aggregation

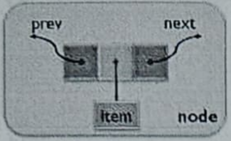
Yield → breaks, and function continues from where it was left off.

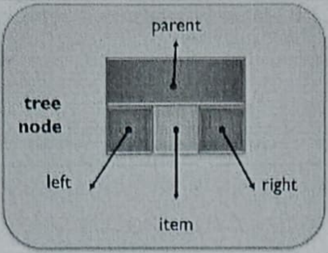
f-strings → used to embed expressions within string literals:

Collection Properties

Collection	Mutable	Duplicates	Ordered	Unique	Iterable	Type hint	Constructor
String	no	yes	yes	no	yes	str	s1: str = ("Python") s2 = str("hello")
Tuple	Items no –Tuple yes	yes	yes	no	yes	tuple	tup1: tuple = (3, "hi", 3.14) tup2 = tuple(("hi",))
Set	Items no –Set yes	no	no	yes	yes	set	set1: set = {3, 7, 11} set2 = set({1, 3, 7})
List	yes	yes	yes	no	yes	list	list1: list = [3, "hi", 3.14] list2 = list([3, 7, 11])
Dict	yes	no	yes	Keys	yes	dict	dict1: dict = {3: "T"} dict2 = dict({, 3: "T", 3: "E"})

Deque	yes	yes	yes	no	yes	deque	from collections
							import deque
							deq2 = deque([3, "hi")





Change 7 to 9 using tail

- tail->next->item->info = 9;
- tail->prev->prev->item->info = 9;

Change 3 to 2 using tail

- tail->prev->item->info = 2;
- tail->next->next->item->info = 2;

Change 5 to 1 using head

- head->next->next->item->info = 1;
- head->prev->item->info = 1;

← Circular Linked list

Change 5 to 6 using tail

- tail->item->info = 6;
- tail->next->next->next->item->info = 6;

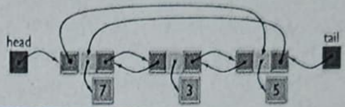
```
typedef int Info;
typedef struct {
      Info info;
} Item;
typedef Item* ItemRef;

typedef struct NodeStruct* NodeRef;
typedef struct NodeStruct {
      ItemRef item;
      NodeRef next;
      NodeRef prev;
} Node;
```

Doubly-linked list

```
typedef struct NodeStruct* NodeRef;
typedef struct NodeStruct {
      NodeRef parent;
      NodeRef left;
      NodeRef right;
      ItemRef item;
} Node;
```

Tree



structs as parameters

structs are passed by value

- □ Entire struct is copied when passed; inefficient
- □ struct logical error; information is lost after call

```
void initDateByValue(Date d) {
    d. day = 19;
    d.month = 11;
    d.year = 1999;
} //initDateByValue
```

structs are usually passed by reference

- □ Date information is retained after the call

```
void initDateByReference(Date* d) {
    d->day = 19;
    d->month = 11;
    d->year = 1999;
} //initDateByReference
```

```
Date: 99/99/99
Date: 99/99/99
Date: 99/99/99
Date: 19/11/1999
```

```
typedef struct {
    int day, month, year;
} Date;

void printDate(Date d) {
    printf("Date: %d/%d/%d\n"
    d.day, d.month, d.year);
} //printDate

int main(void) {
    Date bd = {99, 99, 99};
    printDate(bd);
    initDateByValue(bd);
    printDate(bd);
    initDateByReference(&bd);
    printDate(bd);
    return EXIT_SUCCESS;
} //main
```

```
void* emalloc ( int size ) {
    void* x = malloc ( size );
    if (x == NULL) {
        printf( stderr, "malloc failure" );
        exit ( EXIT_FAILURE );
    return x;
}
```

Use case	Python Code
Every element	no slice, or [:] for a copy
Every second element	[::2] (even) or [1::2] (odd)
Every element but the first one	[1:]
Every element but the last one	[:-1]
Every element but the first and the last one	[1:-1]
Every element in reverse order	[::-1]
Every element but the first and the last one in reverse order	[-2:0:-1]
Every second element but the first and the last one in reverse order	[-2:0:-2]

Slicing

Python m
rando

random.random()

- Generate random float in range [0.0 .. 1.0)

random.randrange(a, b)

- Generate random int in range [a .. b) — [a .. b-1]

b is not included in the random numbers that get generated

random.randint(a, b)

- Generate random int in range [a .. b]

← Random module

seed(k)

- Uses k as seed ← PRNG

seed()

- Uses system clock as seed ← Different seed: sequence is different time

Dynamic
Storage Allocatic

malloc() and free() are declared in <stdlib.h>

```
#include <stdlib.h>
typedef int* IntRef;
IntRef x;
x = (IntRef) malloc ( sizeof ( int ) );
if (x == NULL) exit ( EXIT_FAILURE );
...
free ( ( void* )x );
```

Pointer to allocated memory

Allocates memory on heap

Amount of memory to be allocated

Cast from void* to IntRef pointer

De-allocates memory

Cast from IntRef to void* pointer

→ How malloc() works.

```
from typing import NamedTuple

class Person(NamedTuple):
    name: str
    age: int
    height: float
    country: str = "Canada"

p1: Person = Person(name="Taylor Swift", age=33, \
                    height=180, country="USA")
print(p1)
```

Example of named Tuple