Rose's Programming in Python

Rose Enos

2024

Adapted from the following sources:

- The Python 3.12.1 documentation by the Python Software Foundation
- Notes by Professor Alex Thornton at the University of California, Irvine for I&C SCI 32A and I&C SCI 33
- Lectures by Professor Mustafa Ibrahim at the University of California, Irvine for I&C SCI 32
- Lectures by Cora Schallock at the University of California, Irvine for I&C SCI 32

Contents

1 Program Structures

1.1 The Interpreter

After installation, Python can be executed by the shell command

```
python option
```

with `option`

- None or `-`: enter interactive mode; `sys.argv[0]==""` or `sys.argv[0]=="-"`.
- `script`: execute a `script`; `sys.argv[0]==script`.
- `-c command`: execute a `command`; `sys.argv[0]=="-c"`.
- `-m module`: execute a `module`; `sys.argv[0]==module`.
- `-i module`: execute a `module` and enter interactive mode; `sys.argv[0]==module`.

Interactive mode accepts commands for immediate interpretation. If the command is or returns a statement, the statement is printed and stored in the `str _`. Extraneous arguments are stored as `str` in the `list sys.argv`.

A **comment** is uninterpreted text delimited by `#` and a new line anywhere outside of a `str`. The encoding of a file can be set at the first line:

```
# -*- coding: encoding -*-
```

where `encoding` is a codec.

1.2 Control Structures

Any number of `args` can be printed

```
print(args, end=string)
```

separated by spaces, where `string` ends the output and defaults to a new line.

A **while loop**

```
while condition:
    commands
```

executes `commands` while `condition` is true. A **control structure** controls what is executed. A common control structure is the **if statement** `if <condition>:`. The if statement evaluates the **truthiness** of the condition expression. Truthiness is defined per type.

`object_1 is object_2` returns as a `bool` whether `object_1` and `object_2` are the same object.

1.3 Functions

A **function** `def <function>([params]):` is a defined set of code that can be **called** `function(args)` to execute. With multiple parameters, arguments can be passed in order (positional arguments) or by keyword `name = value` (keyword arguments). Keyword arguments must appear after all positional arguments.

`def function(param = value):` defines a **default argument**. Default arguments must appear after all non-defaulted parameters. Default arguments are stored in the `__defaults__` attribute of the function.

A **tuple-packing parameter** is a `*parameter` that packs any number of arguments into a tuple. At most one tuple-packing parameter can be defined. The tuple-packing parameter defaults to an empty tuple and cannot be manually defaulted.

A **keyword-only parameter** is a parameter that appears after `*` in the parameters and that can only be filled by a keyword argument. Parameters that appear after a tuple-packing parameter are also keyword-only parameters.

A **positional-only parameter** is a parameter that appears before `/` in the parameters and that can only be filled by a positional argument. The keyword-only marker must appear after the positional-only marker if both are present.

A **dictionary-packing parameter** is a parameter `**kwargs` that packs any number of keyword arguments that do not fill other parameters into a dictionary. At most one dictionary-packing parameter can be defined and it must be the last parameter. The dictionary-packing parameter defaults to an empty dictionary and cannot be manually defaulted.

Optional **type annotations** indicate the intended types of variables related to functions:

- `<param>: <type>` the intended argument type.
- `def <function>([params]) -> <type>:` the intended return type.

The **typing library** `from typing import <Type>` allows type annotations for types with the same syntax like `List[...]` and `Tuple[...]` and general types like `Iterable[...]`.

Abstraction is the compartmentalization of code into self-descriptive functions. Abstraction allows easier maintenance based on intended functionality.

1.4 Functional Programming

Functional programming is an alternative to object-oriented programming.

- Functions are the main organizational blocks.
- Functions are **pure**, meaning their result is uniquely determined by their arguments and they have no side effects.
- Functions are **first class**, meaning they are data types.

The benefits of functional programming are

- Clarity: functions are independent except where explicitly stated.
- Testability: functions can be tested independently.
- Provability: mathematics can prove functionality.
- Parallelizability: functions can execute simultaneously where independent.

A **lambda expression** returns an unnamed **anonymous function** `lambda param1, ..., paramn: retur`
A **higher-order function** takes a function as a parameter or returns a function. A higher-order function can return the **composition** of two argument functions, or a **pipeline** of many argument functions. A **partial call** provides fewer than all required arguments to a function. A higher-order function can return a partial function that returns a value when the remaining arguments are passed to it.

An object is **callable** if it can be called like a function. `callable(object)` returns whether an object is callable. Presence of `__call__(self, param1, ..., paramn)` determines whether an object is callable.

Common higher-order functions are

- `map(function, iter1, ..., itern)` returns function value on respective elements of each iterable, with the number of iterables the number of function parameters.
- `filter(boolfunction, iter)` returns values for which the function is true.
- `functools.reduce(function2param, iter, [initial])` returns a value resulting from a function on each element of an iterable (nonempty if the initial value is not present), with the first argument the value resulting from all previous elements (and the initial value, if present) and the second argument the current element.
- `functools.partial(function, arg1, ..., argp)` returns a partial function resulting from passing arguments to the original function.

The `operator` module gives operators as functions

- `operator.add`
- `operator.mul`
- `operator.truth`

1.5 Decorators

A **decorator** is a callable object with a callable parameter that returns a callable object. Writing `@decorator` right above a callable object definition transforms the object into its return value by the decorator. Multiple decorators are applied in reverse order. To pass arguments to a decorator, we can make the decorator a call to a function that accepts parameters and returns a decorator. `getattr(object, name)` returns, and `setattr(object, name, value)` sets, respectively, the attribute of an object from its name as a string.

A **bound method** is a method whose `self` parameter is predetermined. A function is a descriptor and is a bound method if obtained as an attribute of an object. A decorator can be used on a class method by having the decorator return a callable descriptor whose value is the bound decorated method if obtained as an attribute.

Caching is storing data for reuse. Caching function results is memoization. The `functools` module provides the decorator `functools.cache` that implements caching for a function if all the arguments are hashable and the function is pure. This trades off memory complexity for time complexity, and is beneficial when dictionary lookup time is less than the function time and when the dictionary is not prohibitively large. `functools.lru_cache` implements `functools.cache` and removes the least-recently used cached value if a size boundary is hit.

`functools.total_ordering` decorates a class that has `__eq__` and `__lt__` and implements `__ne__`, `__ge__`, `__le__`, and `__gt__` automatically.

1.6 Context Managers

Automatic wrap-up is automatic execution after a certain operation. The `finally` block is automatic wrap-up to the `try` block.

A **context manager** is an object that automates operations based on context. `File`, `socket`, the return from `urllib.request.urlopen(url)`, and the return from `self.assertRaises(error)` are context managers.

`with` is a syntactic sugar because it makes programming easier. In `with context_expression as name`, `context_expression` returns the context manager and stores it optionally in `name`. The context manager is notified of exit from the `with` block. A context expression that does not return a context manager results in `TypeError`.

The `contextlib` module in the standard library provides context managers. `with contextlib.redirect_stdout(io.StringIO()) as output` gives a context manager `output` where `print(string)` sets the value of `output` instead of printing to the shell and `output.getvalue()` returns the value.

A **protocol** determines how the attributes of a class or object are treated. The initialization protocol passes arguments in an object constructor to `__init__`. The context management protocol, or context manager protocol, determines how to enter and exit a `with` block:

- `__enter__(self)` is called as the `with` block is entered and returns the context manager, usually `self`.

- `__exit__(self, exc_type, exc_value, exc_traceback)` is called as the `with` block is exited. If the block is exited normally, the last three parameters are `None`. If an exception is raised, they are the exception type, error message, and traceback, respectively, and returning `True` suppresses the exception.

1.7 Exceptions

An **exception** is an object of type `Exception` that is **raised**, or **thrown**, when an exception condition is met. An exception is identified by its **name**, describing why it was raised, and **trace**, describing where it was raised. A program raises `AssertionError` if an assertion fails, `TypeError` if an operation fails because of the types of its inputs, and `ValueError` if an operation fails because of the values of its inputs.

An exception can be raised manually `raise <exception>`. **Exception handling**, or **exception catching**, is the functionality of a control structure with raised errors as its conditions. A **try block** `try:` executes code until an exception is raised. An **except block** `except <exception> as e:` executes code if a specific exception is raised in the try block and stores the exception in `e`. Exceptions are handled by except blocks in vertical order. An optional **finally block** `finally:` executes code after the executed try blocks, if any. A custom exception

```
class <exception>(Exception):
    pass
```

can also be manually raised. The **pass statement** `pass` ends a block.

1.8 Testing

Unit testing is the evaluation of an individual unit of a program, such as a function or group of related functions. Unit testing is performed with the structure of the unit in mind. **White box testing** is the evaluation of a full program with the structure of the program in mind. **Black box testing** is the evaluation of a full program without knowing how it functions.

A **normal case**, or **happy path**, is a set of tested conditions that simulate the standard intended functionality of the program. An **error case** is a set of tested conditions that are expected to raise an error. A **boundary case**, or **edge case**, is a set of tested conditions that lie on the boundary of standard functionality and error.

The **unit test library** `import unittest` automates unit testing. Tests execute in alphabetical order. The number of passes, number of failures, and run time are displayed after the tests complete.

The **main test function** `main()` begins testing all classes in the program that are extensions of `TestCase`. The **setup method** `def setUp(self):` is called before each test. The **teardown method** `def tearDown(self):` is called after each test.

Tests are methods with names prefixed by `test_`. Tests pass or fail by **assertion statements** which are methods of `self`. Some assertion statements are as follows:

- `assert <statement>` the statement is true.
- `assertEqual(<value 1>, <value 2>)` the two values are equal.
- `with assertRaises(<exception>)`: the code in the block raises the exception.

`fail(message)` fails the test and prints a message.

2 Object Oriented Programming

2.1 Namespaces

A **module** is a `.py` file. The attributes of a `module` are stored in `__dict__`. The attributes of an `object` are stored in `object.__dict__`. The **namespace** of a component is the set of names defined in the component.

The **scope** of a namespace is the largest component on which it is recognized. The **local scope** contains names defined within a block. The **global scope** `global <declaration>` contains names defined within a program. A global variable can be accessed in a lower scope `global <variable>`. The **standard library** contains names defined by Python. `NameError` is raised when a specified name is not defined.

The namespace of a module can be **imported** into another module `[from <module>] import <module/name> [as <alias>]`. To prevent code in a module from running when it is imported, stray code should be placed in the control structure `if __name__ == '__main__':` which executes only if the current module is the main module being executed by the program.

Visibility determines what parts of a module should be visible in which scopes. An **externally available** name is one that is imported from an outside module. An **internally available** name is one that is defined in the current module. A **public** name is one that can be imported to an outside module. A **private** name `__<name>` is one that should not be used in outside modules.

Dunders are internal Python variables delimited by double underlines. `dir()` returns the `list` of dunders.

A **package** is a directory that contains modules.

- `import package.module` imports a module from a package
- `import package` imports the package as specified by `__init__.py`:

```
from .module import name
```

2.2 Classes

Object-oriented programming, or **OOP**, is the practice of programming with **objects**, which are structural models of real objects, instantiated from **classes**, which define the models. OOP allows more abstraction and reusability than procedural programming.

A **class** `class <class>:` defines **fields**, or **instance variables**, which are variables of an object, and **methods**, which are functions of an object. The `__dict__` of a class is a `mappingproxy`, which is similar to a `dict` but does not support assignment. `__dict__` includes

- `__module__` the name of the module in which the class is defined
- `__doc__` the docstring

- `__annotations__` type annotations on the attributes of the class
- `__dict__`
- `__weakref__`

Fields are attributes of the class and are not automatically attributes of objects.

A class describes objects that share behavior. A **public interface** of a class is a set of its methods and their natural-language descriptions. Multiple classes can be defined in a single module, and they are public.

A **Unified Modeling Language Diagram**, or **UML Diagram**, describes a class as a vertically tripartitioned rectangle with the name in the top box, the fields in the middle box, and the methods in the bottom box.

A method is called by `<object>.<method>([params])` or `class.method(object, args)`. Methods are attributes of classes, not of objects. Methods are defined with the first parameter `self` which represents the current object but is not passed in a method call. Instance variables are private. A **getter**, or **accessor**, is a method that returns an instance variable. A **setter**, or **mutator**, is a method that modifies an instance variable.

The **constructor**, or **initializer**, `def __init__(self, [params]):` is a special method that initializes the instance variables when an object is created. The constructor is unique to a class. The special methods **equal** and `def __eq__(self, <compared object>):` and **less than** `def __lt__(self, <compared object>):` overload their respective operators.

A **static method** is marked by `@staticmethod` on the preceding line and does not include the `self` parameter. Static methods are called on the class, not the object. A **class method** is marked by `@classmethod` on the preceding line and takes the `cls` parameter instead of the `self` parameter. A **factory method** is a class method that creates an object of type `cls`.

2.3 Inheritance

Inheritance places the attributes of a **base class**, or **superclass** or **supertype**, under a **derived class**, or **subclass** or **subtype**. In **single inheritance**, a derived class has one base class. A class is automatically a derived class of the `object` class. The `object` class is the only class that is not a derived class. The base classes of a derived class are stored in the tuple `DerivedClass.__bases__`. If an attribute is not implemented in a class, a call defers to the attribute of the base class. The **Liskov substitution principle** states that an object should be able to substitute for objects of any of its base classes.

The base class can be specified by `class DerivedClass(BaseClass).` `type(a) is C` returns whether an object is of a specific class. `isinstance(a, C)` returns whether an object is of a specific class or any of its base classes.

`super(type=type(self), obj_or_type=None)` is an object that represents an object of a base class. The super object can be a parent or sibling class of `type`. The object searches the MRO of the MRO to be searched. The super object begins searching for attributes from the class right after `type`.

In **multiple inheritance**, a derived class has multiple base classes `class DerivedClass(BaseClass1, ...` If an attribute is not implemented in the derived class, the call will defer to the attributes of the base classes in the order they appear in the class definition, which is stored in the **method resolution order**, or **MRO**, tuple `DerivedClass.__mro__` whose first entry is the derived class itself. A **linearization** is an MRO the follows these rules:

- If `X` is a base class of `Y`, then `Y` appears before `X` in the MRO.
- If `X` and `Y` are base classes and `X` is listed before `Y` in the class definition, then `X` appears before `Y` in the MRO.

Python linearizes the MRO by the C3 algorithm. Class definition fails if two base classes have the same base classes in different order.

A **mixin** class is a class that is inherited from and that provides flexible attributes that do not depend on the inheriting class. Mixin classes are usually listed before any base classes in the class definition.

2.4 Class Design

Class design is driven by the intended functionality and prohibitions. We can start with the simplest thing that could possibly work. Then we modify functionality to meet the intention. If the solution is not readable in its current form, we should change it to a more readable form.

`__getattr__(self, name)` returns the value of an attribute that is not defined as part of the object, class, or base classes. `__setattr__(self, name, value)` determines how attribute assignment works. `__delattr__(self, name)` determines how attribute deletion works.

An **attribute descriptor**, or **descriptor**, is an object whose value depends on whether it is obtained as a class attribute. If the object is obtained as a class attribute, it calls `__get__(self, obj, objtype)` whose parameters are the other object, if any, and class from which the object is being obtained, and which returns the value of the object as an attribute. `__set__(self, obj, value)` determines how the attribute is assigned. `__delete__(self, obj)` determines how the attribute is deleted. `__set_name__(self, cls, name)` is called when a descriptor is stored as a class attribute.

A **property** of a class is a descriptor decorated by `@property` that associates a name with values for each object. A no-argument method as a property is its return value. Properties do not support assignment by default. The original method is stored in the property attribute `fget`. The decorator `@propertyname.setter` for some property name determines by the method it decorates how the property supports assignment. `@propertyname.deleter` does the same for deletion.

Method signatures should have type annotations and should have positional-only, keyword-only, or other kinds of parameters as appropriate.

A **dataclass**, provided by the `dataclasses` module, is a class decorated by `@dataclasses.dataclass` that stores values in fields. The fields are uninitialized and have type annotations. An object is initialized with values for

all fields as arguments. We can specify whether fields are mutable by making the decorator a call with the argument `frozen=True` (then the dataclass objects are hashable) and whether they can be initialized by keyword only by `kw_only=True`. Equality is implemented automatically. Objects of a dataclass are usually mutable and so unhashable.

`__post_init__(self)` executes right after `__init__`, if defined.

2.5 Abstract Base Classes

An **abstract base class** describes which methods, called **abstract methods**, a derived class must have. The `collections.abc` module provides abstract base classes. `collections.abc.Sized` describes a sized class. The `numbers` module provides abstract base classes for numeric types. If a derived class does not match a protocol, instantiation raises `TypeError`.

The principle of **goose typing** states that an object is an instance of a protocol if the object implements the necessary methods of the protocol.

The `abc` module provides abstract base classes. `abc.ABC` is the base class for abstract base classes. Abstract methods are decorated by `abc.abstractmethod`. An abstract method raises `NotImplementedError`. An abstract property is an abstract method decorated by `@property`.

A **virtual subclass** of an abstract base class is a class that is considered to implement the abstract base class whether it actually does or not. The `register` class method of an abstract base class takes a virtual subclass as an argument. `@abstractbaseclass.register` decorator does the same thing.

3 Data Structures

3.1 The Python Data Model

The **Python data model** describes how objects can interact with each other, especially by **protocols** relying on the presence of dunders.

- Objects are displayed in the shell by `__repr__(self)` method.
- Objects are initialized by `__init__(self, args)` method with optional arguments.
- `with` statement calls `__enter__(self)` and `__exit__(self, exc_type, exc_value, exc_traceback)`.
- Iteration calls `__iter__(self)` for an iterator that calls `__next__(self)`.
- Object method calls become class method calls with the object as the first argument.

An object is **sized** if it has a length `len(object)` returned by `__len__(self)`. Sized objects are falsy only for length `0`. `__bool__(self)` has priority over length to decide truthiness. An unsized object with no truthiness method is truthy.

An object can be **indexed** by bracket notation `object[index]` if it has `__getitem__(self, index)`. Objects support assignment and deletion if they have `__setitem__(self, index, value)` and `__delitem__(self, index)`, respectively.

An object supports the **sequence protocol** and is an iterable **sequence** if it can be indexed. Sequences are usually sized. Iterable dunders have priority over sequence dunders. Iteration on a sequence ends when indexing raises `IndexError`. `__reversed__(self)` returns an iterator and is usually used for reverse iteration. `__contains__(self, value)` determines `in` keyword functionality.

Slicing returns a subsequence. Slicing with bracket notation `object[start:stop:step]` calls indexing with the index as a `slice(start, stop, step)` object, assigning `None` to omitted indices. Slice indices can be obtained `slice.start`, `slice.stop`, and `slice.step`. Slices are immutable. `slice.indices(length)` returns a tuple of the indices with the stop index modified to make the slice span `length`, with start `0` and step `1` if they are `None` in the slice. Slice assignment and deletion are also supported.

An object is **hashable** if it has a **hash** integer that uniquely describes it, given by `hash(object)` which calls `__hash__(self)`, to be stored in a **hash table**, and it is immutable. A simple hash implementation takes the hash of the tuple of the identifying values of the object. If an object is immutable and has `__eq__`, then it is automatically hashable without implementing `__hash__`.

The **identity** of an object `id(object)` is its unique memory address. Two objects are identical `a is b` if they have the same identity. Two objects are equal `a == b` if they have the same meaning as determined by `__eq__(self, other)`

which returns a Boolean value or `NotImplemented` of `NotImplementedType`, and automatically compares identities if not implemented. Inequality `a != b` negates equality by default, but can be defined by `__ne__(self, other)`.

- If two objects are equivalent, then they have the same hash.
- If two objects have different hashes, then they are not equivalent.

Relational comparison compares objects by an ordering system. **Lexicographical ordering** compares two lists by their respective elements, starting from the first pair until the first unequal pair, or by length if all respective pairs are equal. Alphabetization is lexicographical. We can implement less than `__lt__(self, other)`, greater than `__gt__(self, other)`, at most `__le__(self, other)`, and at least `__ge__(self, other)`. Implementing one direction of an inequality automatically implements the other direction, but implementing a strict inequality does not automatically implement a non-strict inequality.

When comparing two objects of different types, the method of the first object is used if implemented, and otherwise the method of the second object is used.

The unary plus operator `+object` uses `__pos__(self)`. The unary minus operator `-object` uses `__neg__(self)`. Other arithmetic operators are

- `__add__(self, other)` addition
- `__sub__(self, other)` subtraction
- `__mul__(self, other)` multiplication
- `__truediv__(self, other)` division
- `__floordiv__(self, other)` floor division
- `__pow__(self, other)` exponentiation

If an arithmetic operation does not exist or returns `NotImplemented`, the operation raises `TypeError` or defers to a **reflected operator** method of the second operand if it exists.

- `__radd__(self, other)` reverse addition
- `__rsub__(self, other)` reverse subtraction
- `__rmul__(self, other)` reverse multiplication
- `__rtruediv__(self, other)` reverse division
- `__rfloordiv__(self, other)` reverse floor division
- `__rpow__(self, other)` reverse exponentiation

An **augmented arithmetic operator** modifies an existing object. Augmenting an immutable object returns a copy. Augmented arithmetic is implemented automatically based on arithmetic methods, or can defer to augmented arithmetic methods.

- `__iadd__(self, other)` in-place addition
- `__isub__(self, other)` in-place subtraction
- `__imul__(self, other)` in-place multiplication
- `__itruediv__(self, other)` in-place division
- `__ifloordiv__(self, other)` in-place floor division
- `__ipow__(self, other)` in-place exponentiation

`dict` and `set` have a **union** operation `a | b` that returns the same type object with the union of the values of each component.

- `__or__(self, other)`
- `__ror__(self, other)`
- `__ior__(self, other)`

3.2 Types

The **type** of a variable is its format. A **data structure** is a type that stores multiple pieces of data. The type of a `value` can be obtained

```
type(value)
```

A `value` can be **cast**, or **typecast**, to a `new_type`

```
new_type(value)
```

A value can be **cast**, or **typecast**, to a different type as an argument of the function whose name is the target type. For example, `int(<value>)` casts a value to an `int`, and `str(<value>)` casts a value to a `str`.

3.3 Booleans

A `bool` is a Boolean value. The `int 0` or any value of length `0` is `False`. Any nonzero `int` or any value of length at least `1` is `True`.

Delimiters	`()`
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Float division	`/`
Floor division	`//`
Modulo	`%`
Exponentiation	`**`

Table 1: Arithmetic operators

Less than	`<`
Greater than	`>`
Equal to	`==`
Less than or equal to	`<=`
Greater than or equal to	`>=`
Not equal to	`!=`

Table 2: Comparison operators

3.4 Numbers

An `int` is an **integer**. A `float` is a **floating-point number**. In arithmetic, `float` dominates `int`.

The **assignment** operation

```
variable_1, ..., value_n = value_1, ..., value_n
```

sets the `value_i` of a `variable_i`. By the principle of **duck typing**, the type of a variable is the type of its value. A variable without a value does not exist. The names of variables should follow exactly one convention per project. Some common Python conventions are **PEP 8** and the **Google Style Guide**.

The **math library** provides mathematical functions including the following:

- Square root `sqrt(<number>)`
- Truncation `trunc(<number>)`
- Cosine `cos(<number>)`
- Sine `sin(<number>)`
- Tangent `tan(<number>)`
- Euler exponential `exp(<number>)`
- Conversion to degrees `degrees(<angle in radians>)`
- Conversion to radians `radians(<angle in degrees>)`
- Logarithm `log(<exponential>, <base>)`

3.5 Strings

A `str` is a **string**. A **string literal** is the definition of a `str` delimited by `""` or `''` on one line, or `"""` and `"""` or `'''` and `'''` on several lines. A **docstring** is a description of a class or method delimited by triple quotes, and is stored in `__doc__`.

The **concatenation** operation `+` joins two `str`. String literals separated only by whitespace are automatically concatenated. The **repetition** operation `*` repeats a `str` an `int` number of times. **Overloading** is the multiple definition of operators based on the types of their inputs.

The **escape character** \ immediately precedes special or escaped characters. A **raw string** is a string literal immediately preceded by `r` and does not interpret special characters. A **byte literal** is a string literal immediately preceded by `b` and is of type `bytes`.

Empty character	New line
Backslash	\
Single quotation	'
Double quotation	"
New line	`n`
Tab	`t`

Table 3: Special and escaped characters

A character can be accessed

```
string[index]
```

by its `index`, from the start if positive and from the end if negative, in a `string`. Using an invalid index raises `IndexError`. A substring can be sliced

```
string[start:end:step]
```

including the optional `start` index and excluding the optional `end` index, moving by `step`. Using invalid indices raises no error. A `str` is immutable. The length of a `string` is `len(string)`.

The **lowercase function** `<str>.lower()` returns a string whose characters are all lowercase. The **uppercase function** `<str>.upper()` returns a string whose characters are all uppercase. The **replacement function** `<str>.replace(<old>, <new>)` returns a string in which all instances of an old string are replaced with a new string.

3.6 Iterables

A `list` is a **list** delimited by `[]` of other values separated by `,`. A `list` supports indexing, slicing, concatenation, and length. A `list` is mutable. An `item` can be appended to a `list`

```
list.append(item)
```

An **iterable** is a data structure whose values can be accessed in sequence. Some common iterables are `str`, `range`, `list`, `dict`, `set`, and `tuple`. `list(iterable)` creates a list containing each element in an iterable. The **map function** `map(<function>, <iterables>)` calls a function for each value in as many iterables as arguments required by the function. `max(iterable)` returns the maximum element of the iterable.

The values in an iterable can be assigned to individual variables by **sequence assignment**

```
x_1, ..., x_n = iterable_of_length_n
```

The values in an iterable can be **unpacked** by `*iterable` for usage as individual values, such as arguments to a function. The values in a mapping can be **dictionary unpacked** by `**mapping`.

Due to hardware advancements, data structures usually prioritize performance over storage efficiency. A `list` is ordered and mutable, and can contain duplicates.

A `dict` is ordered by keys, with keys immutable and values mutable, and cannot contain duplicates. A list of keys of a `dictionary` is given by

```
dictionary.keys()
```

The **hash function** `hash(<value>)` returns an integer representing an immutable object. The hashes of unique identifiers of values can be used as keys in a `dict` because the hashes of unique objects are unique.

The memory location of an immutable variable changes when the variable value changes because the variable must point to a modified copy of the original value. The memory location of a mutable variable does not change when the variable value changes.

A set cannot contain duplicates or mutable elements.

3.7 Comprehensions

A **comprehension** is an expression that builds a data structure. A **list comprehension** builds a list based on a description of values.

```
[list_element
for element_1 in iterable_1 . . . for element_n in iterable_n
if condition_1 . . . if condition_m]
```

builds a list of `list_element` created from `element_i` of `n` `iterable_i` that meet `m` optional `condition_i`.

A **set comprehension** builds a set with `{}` delimiters instead of `[]`. Duplicate elements are not added to the set. Mutable elements raise `TypeError` when trying to be added to the set. A set cannot contain duplicates or mutable elements. `set.add(element)` adds `element` to a `set`.

Hashing creates a unique integer (**hash value**, or **hash**) that describes an object. The hash determines where the object is stored. A **hash table** stores hashes. Hashable objects must be immutable.

- `__hash__(self)` creates a hash for the object.
- `__eq__(self, other)` determines whether the object is equal to another object.

A **dictionary comprehension** builds a dictionary with `{}` delimiters and `key: value` instead of `list_element`. The keys must be hashable but the values do not need to be hashable. Dictionaries can be looped through with `for key, value in dictionary`.

A **generator comprehension** builds a generator with `()` delimiters. `tuple(generator)` returns a tuple of the elements in `generator`.

A simple list comprehension using a range runs in $O(n)$ time with $O(n)$ resource complexity. A `range(start, stop, step)` stores only `start`, `stop`, `step`, and the last element returned, so a range has resource complexity $O(1)$. A list stores only the memory location of the first element, the length, the size of a reference, and its references, so it has $O(n)$ resource complexity. Adding n elements to the end of a list creates n new memory location and runs in $O(n)$ time. Adding n elements each to the beginning of the list creates n new memory locations and moves the existing elements toward the end at each addition, so it runs in $O(n^2)$ time.

3.8 Databases

A **database** is a collection of data that is managed by programs over time. A **database management system**, or **DBMS**, is software that manages a database. A **relational database** explicitly relates data to each other. A **table** is a data structure with **rows** and **columns**. A table stores one kind of data. A row shows a datum. A column shows an attribute of the data. The same attributes of different data are the same type. Rows should be unique. Attributes should be scalars, in that they are in the smallest meaningful form.

A **primary key** is a subset of attributes that is unique for each datum, specified explicitly. A DBMS can check that a primary key is unique and associate a primary key with its datum to improve search complexity. An **index** is an ancillary data structure that has two attributes: a primary key and a row number. A primary key should be static so that it does not have to be changed in all of its relevant tables and indices.

A **relationship** indicates how data are related to each other in a database. Relationships use **foreign keys**, primary keys from other tables, as references. A DBMS can enforce **referential integrity** to make sure foreign keys actually exist as primary keys.

The **cardinality** of a relationship is how many elements are related. A one-to-one relationship relates exactly one element from each table to each other, ensuring each foreign key is unique. A one-to-many relationship relates one element to many other elements but the other elements to only that one element, not ensuring foreign keys are unique. A many-to-many relationship relates many elements from each table to each other. Appropriate cardinality depends on the problem, making another table to hold the relationships between keys.

3.9 SQLite

The module `sqlite3` is an embedded DBMS. A database can be persistent or temporary. `sqlite3.connect(path)` returns a database connection `connection`, where `path` is a string either `":memory:"` or a path to a database file. Executing a **statement** interacts with the database in a certain way. Statements are written in **Structured Query Language**, or **SQL**.

The statement

```
CREATE TABLE table(
    primary_key TYPE1 PRIMARY KEY,
    attribute TYPE2
) STRICT;
```

creates a table `table` with primary key `primary_key` of type `TYPE1` with attribute `attribute` of type `TYPE2`, where `STRICT` means values must be of the attribute type. Keywords are conventionally in upper case, whitespace has no meaning, and statements are conventionally ended by a semicolon. A statement `statement` as a string can be executed by `connection.execute(statement)`, which returns a `Cursor` that contains any data returned by the statement.

Metadata is data that describes the database. A **query** accesses data from the database.

```
SELECT column FROM table;
```

returns the `column` attribute of every row in table `table`.

```
SELECT name FROM sqlite_schema;
```

returns the names of all tables in the database. Then `cursor.fetchone()` returns one row as a tuple and moves the pointer to the next row, or `None` when there are no more rows. We should close the cursor when finished `cursor.close()`.

Appending `WHERE condition` affects only rows that meet the `condition`. Equality and order are handled by `=`, `<`, `>`, `<>` (not equal). `AND` is the conjunction. `BETWEEN a AND b` is the range between `a` and `b`. Appending `ORBER BY attribute order` sorts the results by an attribute `attribute` or other feature such as the length `length(attribute)` in the `order` either `ASC` or `DESC`.

```
INSERT INTO table (attribute1, ..., attributen)
VALUES (value1, ..., valuen);
```

inserts a row into table `table` with respective values `valuei` to attributes `attributei`.

```
UPDATE table
SET operation
WHERE condition;
```

updates a row that meets `condition` by `operation`. Assignment is done by `=`.

```
DELETE
FROM table
WHERE condition;
```

removes the rows that meet `condition`.

```
DROP TABLE table;
```

deletes the table `table`.

Missing data are `NULL` and can be selected in a condition by `attribute IS NULL` or `attribute IS NOT NULL`. In the creation of a table, appending `NOT NULL` to an attribute prevents an attribute from being `NULL`. Appending `CHECK (condition)` forces the value to meet the `condition`. `UNIQUE` forces each value to be unique in the attribute. `PRIMARY KEY` is the last constraint. Constraints can be put on the entire table by listing them like attributes `UNIQUE (attribute1, ..., attributen)`, `CHECK (condition)`. Constraints control **data integrity**.

```
CREATE TABLE relationship(
    key1 INTEGER NOT NULL,
    key2 INTEGER NOT NULL,
    PRIMARY KEY (key1, key2),
    FOREIGN KEY key1 REFERENCES table1(key1),
    FOREIGN KEY key2 REFERENCES table2(key2)
) STRICT;
```

makes a many-to-many relationship table with foreign keys, where $(key1, key2)$ uniquely identifies each row in the table. Queries can be written with **joins** that combine rows from different tables by a **join condition**.

```
SELECT a.attribute
FROM tablea AS a
    INNER JOIN tableb AS b ON b.key1 = a.key1
    INNER JOIN tablec AS c ON c.key2 = a.key2
WHERE c.key2 = value;
```

An **injection attack** injects code into input other than what is intended to execute. A **parameterized** statement has placeholders. `connection.execute(statement, arguments)` inserts **positional** `arguments` as a tuple into `statement` which has `?` as each parameter, or **named** `arguments` as a dictionary into `statement` which has `:param` as each `param` matching the dictionary keys.

A **transaction** is a sequence of statements that either executes entirely or does not execute. `connection.commit()` commits all previous executed statements as a transaction. Multiple transactions can read at once but writing transactions are queued. After a timeout in the queue, a transaction raises `sqlite3.Error` with `sqlite3.Error.errorcode == sqlite3.SQLITE_BUSY`.

4 Algorithms

4.1 Iteration

An **iterable** is an iterable object. An **iterator** facilitates iteration. `iter(iterable)` returns an iterator for the `iterable`. `next(iterator)` returns the next value from the `iterator` and raises `StopIteration` on no more elements. The iterable protocol requires

- `__iter__(self)` returns the associated iterator

The iterator protocol requires

- `__next__(self)` returns the next element and raises `StopIteration`
- `__iter__(self)` returns the iterator

`__repr__(self)` determines how the object is represented in the shell, usually `<objecttype object at memorylocation>`. `print(iterator, sep=separator)` prints the elements with `separator` between them. Iterators have $O(1)$ resource complexity and $O(n)$ time complexity.

A **generator** is a function that returns a sequence of results. A **generator function** with parameters `start, end` is a function that returns a generator and manages its iteration from `start` to `end`. `yield value` makes a generator function and adds the `value` to the generator, stops the iteration, and directly precedes the start of the next iteration. The call to a generator function only returns an associated generator. Iteration on the generator executes the code in the generator function and ends with the function. Returning a value raises `StopIteration` with `StopIteration.value == returnedvalue`.

A generator comprehension with delimiters `()` returns a generator. Generators use **lazy evaluation** by only using resources when necessary. Generators have $O(1)$ resource complexity and $O(n)$ time complexity. An **infinite generator** returns an infinite sequence. Infinite generators are usable because of lazy evaluation. Nested generators can form a pipeline of curated results. `yield from generator` iterates and returns every element in the `generator`. Built-in iterating functions are

- `map(func, iter)` returns an iterator of the returned values on the elements
- `filter(boolfunc, iter)` returns an iterator of the truthy-evaluated elements
- `any(iter)` returns true if any elements are truthy
- `any(generatorcomp)`
- `all(iter)` returns true if all elements are truthy
- `all(generatorcomp)`

- `enumerate(iter)` returns an iterator of tuples whose first element is an index and whose second element is from the iterable
- `zip(iter1, ..., itern)` returns an iterator of tuples whose elements are from the same index of each iterable

The `itertools` module provides

- `islice(iter, <end | start, end>)` returns the elements indicated by the nonnegative indices
- `count(start)` returns an infinite sequence of consecutive integers from the start index
- `repeat(value)` returns an infinite sequence of repetitions of the value
- `chain(iter1, ..., itern)` returns an iterator that concatenates the arguments

4.2 Recursion

Loops always execute at least once. **Iteration** is repetition of code until a condition is met. **Recursion** is the process of a function calling itself on a smaller input in a **recursive case** until it reaches a **base case**. A **runtime stack** is a set of pending executions.

A **recursive method** is a method that uses recursion. The default maximum stack space is 1024 calls, after which the function raises `RecursionError` in **stack overflow**. Recursive methods can perform search, sort, parse, and factorial algorithms. Recursion is limited in embedded systems software where memory is limited.

Direct recursion is a function calling itself. **Indirect recursion**, or **mutual recursion**, is a function calling another function that eventually calls the original function. **Single recursion** is a function making at most one recursive call. Single, direct recursion has $O(n)$ time complexity and $O(n)$ resource complexity.

Memoization is remembering a previous result on certain arguments to a function and using the stored result instead of recalculating it on other calls with the same arguments. Memoization can be implemented by a list of length the known total number of results on different arguments based on the initial argument, where a nested memo function effects the recursive case by using the stored value in the list or calculating and storing it if it does not yet exist. Memoization can reduce time complexity for repetitive solutions.

An **optimal substructure** is a recursive case that always returns the same value on given arguments, regardless of the broader problem. **Dynamic programming** calculates the necessary memos at each step by iteration and takes the final answer without recursion based on the memos. Dynamic programming can avoid the stack limit for memoized solutions. Depending on the problem, dynamic programming can reduce resource complexity for memoized solutions.

Tail call elimination, or **tail call optimization** or **TCO**, reduces resource complexity to $O(1)$, but is not supported by Python. A **tail call** makes the called function's result the calling function's result. In other words, the calling function does no more work after the call. An **accumulator** is an extra parameter that stores an accumulated value so that a recursive function can use tail calls. Recursion with tail call elimination can be implemented by a nested recursive function within the original recursive function that effects the recursive case, called by the original function passing a hardcoded accumulator.

4.3 Searching

A `list` is composed of a reference to a memory block with references to elements, an integer representing the size, and an integer representing the capacity of the current memory block. Elements in a `list` are stored contiguously. Elements can be accessed in constant time because the reference block can be found in constant time and all references are the same size, so the product of the index and the size of a reference added to the beginning of the reference block gives the reference to the specified element.

Sequential search, or **linear search**, increments through each element of a `list` in order until the specified element is found. The algorithm stores only the current index, so resource complexity is $O(1)$. The algorithm takes the most time when the element is at the end or not present, and there is a constant number of operations per element, so time complexity is $O(n)$.

Binary search takes the relevant half of a sorted `list` repeatedly until the specified element is found. v lists must be treated where there are n objects,

$$n = 2^v$$

so the time complexity is $O(\log n)$. The algorithm must only store the first and last indices of the current portion of the list and the middle index of that list, so resource complexity is $O(1)$.

5 Input and Output

5.1 Files

Input is data passed into an element. Input can be **programmatically generated** by elements of the program or **externally generated** by sources outside the program, like users, files, databases, and the internet.

Output is data passed out of an element. Output can be **programmatically generated** to elements of the program or **externally generated** to vessels outside the program, like the shell.

The **input function** `input([str])` prints an optional `str` and returns user input from the shell. The **print function** `print([str])` sends a `str` and a new line to the shell.

The `File` class of the standard library allows a program to manipulate files. A `File` is an iterable of the lines in the file. The **open function** `open(<file>, [mode], [buffer])` returns a `File` in a certain **access mode** (default `rt`) that determines how the object can manipulate the file and with a **buffer size** (default `1024`) that determines how much data from the file is available to be manipulated. The access modes are as follows:

- Read only `r` (fails if file does not exist)
- Overwrite `w` (creates file if file does not exist)
- Create `x` (fails if file exists)
- Append `a` (creates file if file does not exist)
- Binary `b`
- Text `t`
- Read and write `+`

A file must be opened before it is manipulated. A file must be closed by the **close function** `<file>.close()` to save changes made by the program. The **with block** `with open(<file>, [mode], [buffer]) as <File>:` creates a `File` to be used within the block and does not require a close function.

The **read functions** `<file>.read([buffer])`, `<file>.readline()`, and `<file>.readlines()` return a certain number of bits (default all), or a single line, or all lines as a `list`, respectively, of an open file and move the **pointer**, where a next read function would begin, to the character after the section that has been read. The read functions return the empty string at the end of the file. A file is also an iterator `for line in file`.

5.2 CSV Files

A **comma-separated value file**, or **CSV file**, contains tabular data separated by commas and newlines. The `csv` library `import csv` parses and manipulates CSV files.

A CSV file is opened in a block `with open(<file>, <mode>, newline = <newline character>) as <File>:` with a row separator specified. Then the read function without argument returns a 2-dimensional `list` that contains the entire table by rows.

A **CSV writer** `writer(<File>)` allows the program to write data to the file. The **write row function** `writerow(<list>)` writes a list as a row. A **CSV reader** `reader(<File>)` is an iterable of the rows in the file as `list` objects.

5.3 PathLib and Glob

A **file identifier** is a path and name. A **file system structure** determines the formats of drives and paths.

The `Path` object `from pathlib import Path` can manipulate paths. The **path function** `Path(<str>)` returns a `Path` whose format is independent of the file system structure. Some `Path` methods are as follows:

- `<Path>.exists()` returns whether a `Path` refers to a real path on the system.
- `type(<Path>)` returns the `Path` subclass associated with the file system structure.
- `<Path 1>/<Path 2>` returns the concatenation of two paths.
- `<Path>.is_file()` returns whether the `Path` refers to a file.
- `<Path>.is_dir()` returns whether the `Path` refers to a directory.
- `<Path>.open([mode])` returns a `File` in an access mode.
- `list(<Path>.iterdir())` returns a `list` of `Path` objects in a directory.
- `Path(<path>).glob('*.'.<extension>)` returns all `Path` objects in a directory of a specific file type.
- `Path.cwd()` returns the current working directory.
- `<Path>.rfind(<str>)` returns the index of the first occurrence of a string in the `Path`, starting from the right end.

5.4 Sockets

A **socket** is an endpoint of a bidirectional data stream. Data is received by a socket in the order sent. Sockets can be used by processes, local programs, remote programs, and remote machines.

The **host** is the device that another device connects to. An **Internet Protocol**, or **IP address**, is a unique machine identifier on a network. An IPv4 address contains four integers in 0-255 (8 bits, or 1 byte). An IPv6 address is an

alternate identifier. A **hostname** is a unique machine identifier. The **loopback address** `127.0.0.1` refers to the current machine.

A **port** is a unique program identifier on a system. A port is an integer in 0-65535 (2 bytes). Ports are managed by the **firewall**, which blocks access to a machine unless an allowed port is specified by the connection.

The type of a socket determines the rules of data transfer. **Transmission Control Protocol**, or **TCP**, `SOCK_STREAM` connects a client and a server. A TCP stream **reliable** because of the following properties:

- Detect and re-transmit lost data packets (three-way acknowledgements, ACKS)
- Data received in order sent: sequence numbers for each packet
- Data integrity checks: confirm data received without errors
- Compare checksum: number of bytes in packet

Unit Datagram Protocol, or **UDP**, `SOCK_DGRAM` opens a continuous data stream for use by clients and a server. UDP only performs a client-side checksum comparison. UDP is faster than TCP because it does not maintain reliability. TCP and UDP use **Client-Server Architecture** in which connections are made between a server and a client.

The **connection protocol** determines how a socket identifies the machine to connect to. A socket can use IP `AF_INET` or **Unix Domain** `AP_UNIX`.

Socket methods are as follows:

- `socket(<protocol>, <type>)` returns a `socket`.
- `<socket>.bind((<host>, <port>))` server; associates connection information with socket.
- `<socket>.listen(<queue=0>)` server; starts socket with maximum connection request queue.
- `<socket>.accept()` server; returns `list` of client `socket` and client address.
- `<socket>.connect((<host>, <port>))` client; initiates connection.
- `<socket>.getblocking()` checks whether socket is **blocking**, or preventing execution until it receives data.
- `<socket>.setblocking()` toggles whether socket is blocking.
- `<socket>.recv(<buffer>)` receives message from socket.
- `<socket>.send(<byte-like>)` sends message from socket.
- `<socket>.sendall(<byte-like>)` sends data until all received or error raised.

- `<socket>.close()` closes socket.
- `<socket>.gethostname()` returns hostname.

A **byte-like object** is a binary string. A `str` can be converted to a byte-like object `<str>.encode("ascii")` or `b<str>`. A byte-like object can be converted to a `str` `<str>.decode("ascii")`.

In a **graceful shutdown**, the server closes the client socket before the client closes its socket. A **zombie** is a process that remains open after completing its tasks. An **orphan** is a process that remains open after its parent process has closed.

A **network-time diagram** shows parallel timelines of events on the server and the client.

5.5 Tkinter

A **Graphical User Interface**, or **GUI**, is a visually intuitive user interface. The **Tkinter** library `import tkinter as tk` can be used to create a GUI.

The **Tk function** `Tk()` returns a window for the GUI. A **widget** is a component of the interface. Some window methods are as follows:

- `<window>.title(<str>)` sets the window title.
- `<window>.geometry(<str>)` sets the dimensions of the window in pixels `"<width>x<height>"`.
- `<window>.configure([options])` sets properties of the window.
  - `background = <str>` sets the background color.
- `<window>resizable(<int>, <int>)` sets by `0` or `1` whether the width and height, respectively, can be changed by the user.

The **main loop** `<window>.mainloop()` updates the window infinitely and receives events. The **quit function** `<window>.quit()` ends the main loop. The **destroy function** `<window>.destroy()` ends the main loop and closes the window. An event and corresponding function can be **bound** to a widget `<widget>.bind(<event>, <function>)`. Some events are as follows:

- `"<Enter>"` the enter key.
- `"<Leave>"` the escape key.
- `"<Button-1>"` the left mouse button.
- `"<Button-2>"` the right mouse button.
- `"<Button-3>"` the middle mouse button.
- `"<Double 1>"` the left mouse button twice.

A **Tkinter variable** `<Var>()` returns a variable that can be used by widgets. The value of a variable can be read `<Var>.get()`. The value of a variable can be set `<Var>.set(<value>)`. Some Tkinter variables are `BooleanVar`, `DoubleVar`, `IntVar`, and `StringVar`.

Some widgets are as follows:

- `Label(<window>, [options])` text or image display.
- `Button(<window>, [options])` clickable button.
- `Entry(<window>, [options])` text entry field.

Some widgets in the additional library `from tkinter import ttk` are as follows:

- `Combobox(<window>, <StringVar>, <list>, [options]` dropdown selection.

Some widget options are as follows:

- `text = <str>` text displayed.
- `textvariable = <Var>` where widget content is stored.
- `font = <str>` font and text color.
- `height = <int>` widget height.
- `anchor = <str>` cardinal text alignment.
- `width = <int>` widget width.
- `command = <function>` function called on click.

A **lambda function**, or **anonymous function** or **throwaway function**, `lambda:` is a one-line function that cannot be called by name. Using a lambda function in the `command` option prevents execution of the function on interpretation of the widget.

A **layout manager** organizes widgets on the window. The **pack manager** `<widget>.pack([options])` stacks widgets vertically downward and has the following options:

- `expand` make widget fill parent.
- `fill` make widget fill space.
- `side = <str>` widget alignment.

The **grid manager** `<widget>.grid([options])` places widgets into a grid and has the following options:

- `column = <int>` column from left.

- `columnspan = <int>` number of columns spanned.
- `row = <int>` row from top.
- `rowspan = <int>` number of rows spanned.
- `ipadx = <int>` horizontal padding between cell and widget.
- `ipady = <int>` vertical padding between cell and widget.
- `padx = <int>` horizontal padding between cells.
- `pady = <int>` vertical padding between cells.
- `sticky = <str>` cardinal widget alignment.

5.6 Application Programming Interface

The **Hypertext Transfer Protocol**, or **HTTP** facilitates internet communication. The **HTTP library** `import http` allows communication with a remote machine through the internet. The client is the program and the server is a web server.

Cookies are data stored persistently on the client side. The **cookie jar** is the set of cookies. The **request method** determines how a client connects to a server. Some requests are as follows:

- `GET` retrieve data.
- `HEAD` retrieve header.
- `POST` submit data.
- `PUT` replace data.
- `DELETE` delete data.
- `CONNECT` establish tunnel.
- `OPTIONS` change communication options.
- `TRACE` retrieve connection path.
- `PATCH` partially modify data.

The **header** is the information about the connection returned by the web server. The header includes the date, server, content length, and content type. A **status code** is a code returned in the header that describes the result of the connection. Some status codes are as follows:

- `200` OK.
- `404` not found.

- `403` access denied.

A **Uniform Resource Locator**, or **URL**, `import urllib` is a unique server identifier on the internet. The **requests library** `import requests` allows a program to send requests to a URL. Requests can include a **query string** `?<param>=<value>&...`that specifies what is requested with **parameters**.

The **URL open function** `request.urlopen("http://" + <url>` returns the response to a query. The **session** `from requests import Session` contains persistent data on the client side during a connection. The **session object** `Session()` can contain keys, preferences, and a timeout. Some `Session` methods are as follows:

- `<Session>.get(<url>, params = <params>)` returns the response to a query where the parameters are a `dict`.
- `<Session>.headers.update(<headers>)` sets the headers used in a query where the headers are a `dict`.

JavaScript Object Notation, or **JSON**, is a standardized data format. The **JSON library** `import json` allows a program to parse JSON data. The **parse function** `<str>.json()[<tag>]` returns a JSON string, optionally of only the data under a specific tag.

Data Pretty Printer `from pprint import pprint as pretp` is a library that improves the formatting of JSON strings. The **pretty print function** `pretp(<JSON>)` prints to the shell a formatted JSON string.

An **Application Programming Interface**, or **API**, allows clients to request data from a web server using a query, and usually requires the client to use a key. The key is usually passed as a header or a parameter.