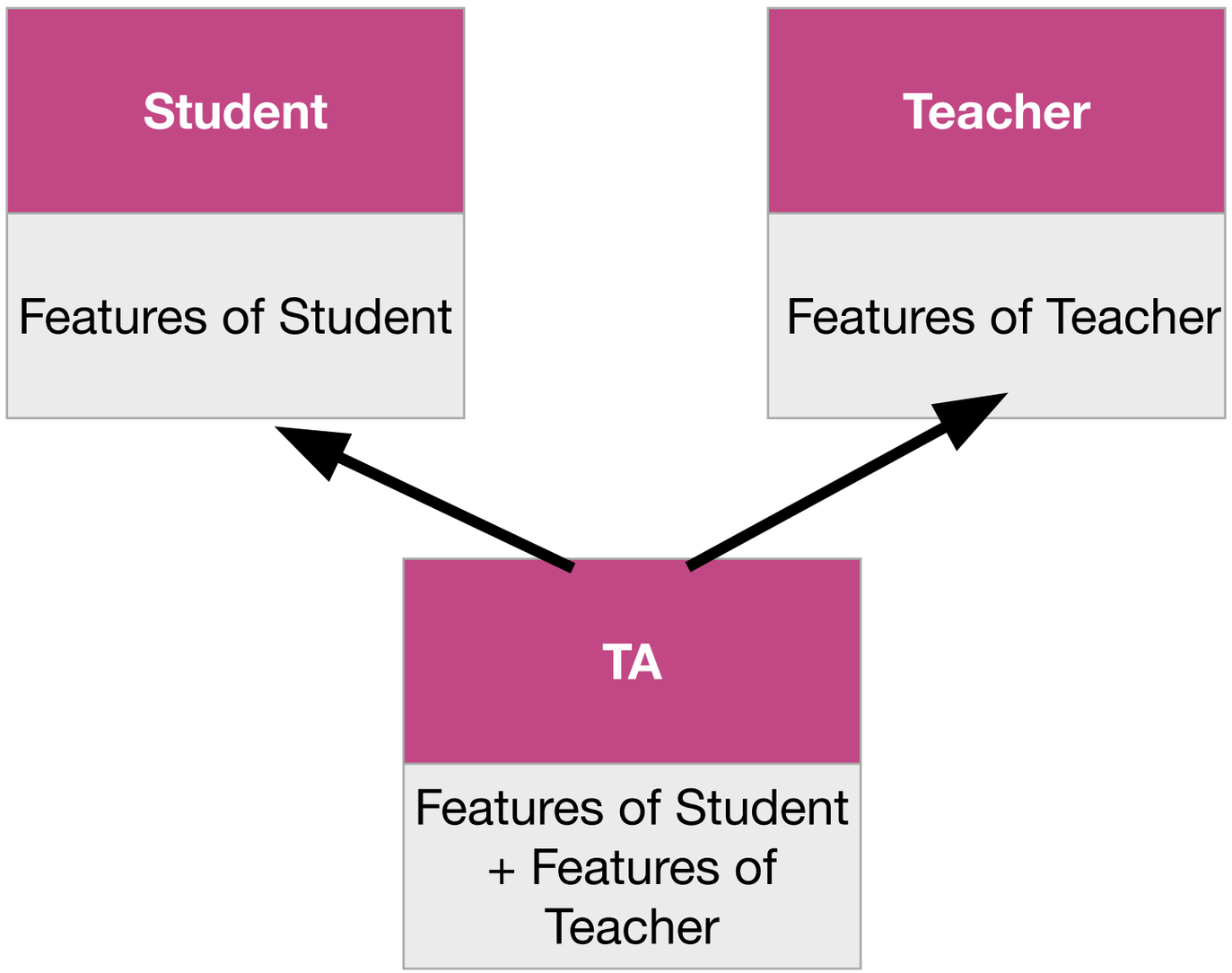
More on Inheritance

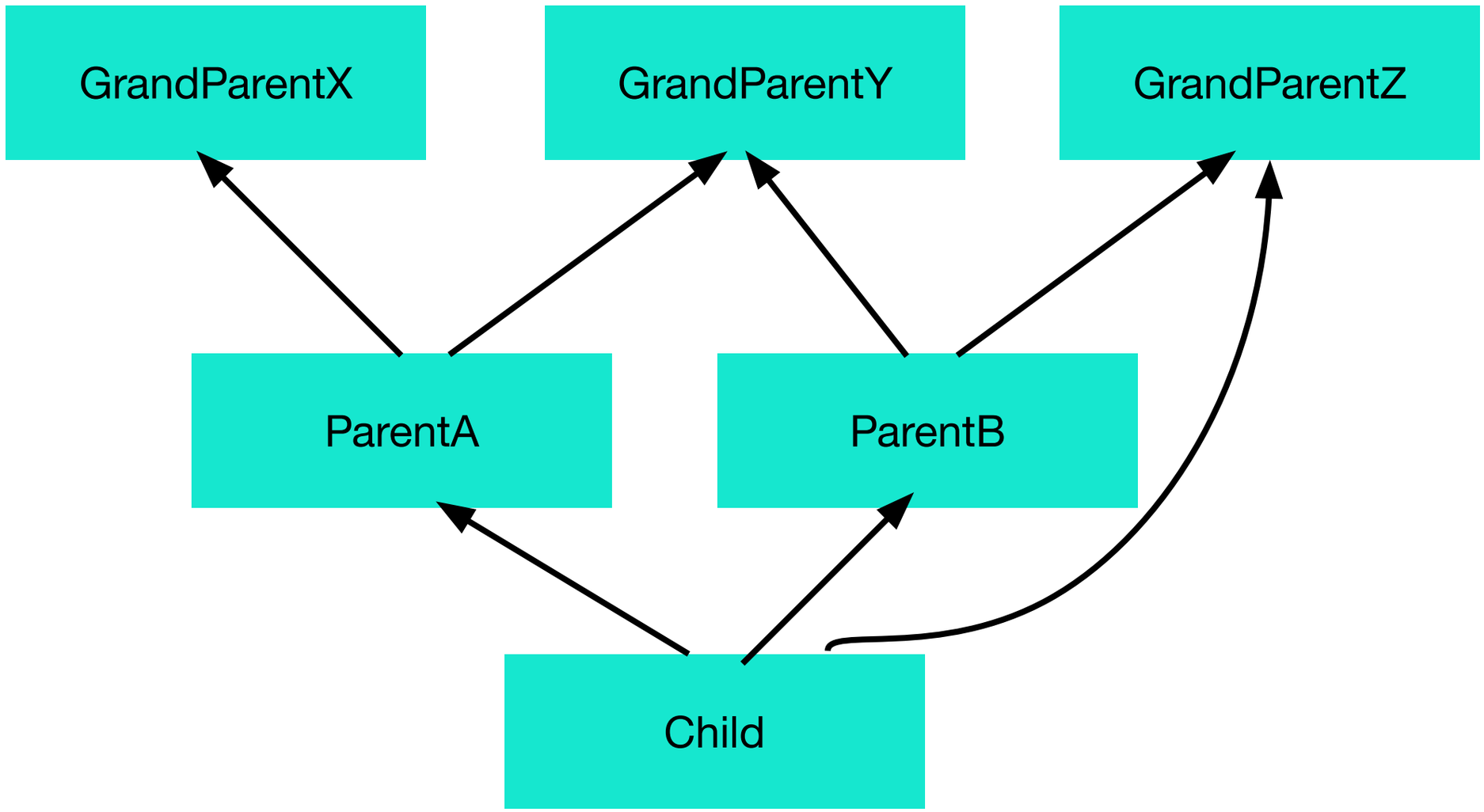
Multiple Inheritance

- A class can be derived from more than one base class in Python, similar to C++. This is called multiple inheritance.
- In multiple inheritance, the features of all the base classes are inherited into the derived class.



Method Resolution Order (MRO)

- In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching the same class twice.
- This order is also called linearization of MultiDerived class and the set of rules used to find this order is called Method Resolution Order (MRO).



Example of Method Resolution Order (MRO)

```
# Demonstration of MRO

class GrandParentX:
    pass

class GrandParentY:
    pass

class GrandParentZ:
    pass

class ParentA(GrandParentX, GrandParentY):
    pass

class ParentB(GrandParentY, GrandParentZ):
    pass

class Child(ParentA, ParentB, GrandParentZ):
    pass


print(Child.mro())

"""
[<class '__main__.Child'>,
<class '__main__.ParentA'>, <class '__main__.GrandParentX'>,
<class '__main__.ParentB'>, <class '__main__.GrandParentY'>,
<class '__main__.GrandParentZ'>,
<class 'object'>]
"""
```

Problem of Multiple Inheritance - The left parent class (Teacher) was selected first for __init__(), dowork(), and display().

```
class Student:
        def __init__(self, name, gpa):
                self.name = name
                self.gpa = gpa

        def dowork(self):
                print('Student', self.name, 'doing homework.')

        def display(self):
                print('name =', self.name)
                print('gpa =', self.gpa)

class Teacher:
        def __init__(self, name, salary):
                self.name = name
                self.salary = salary

        def dowork(self):
                print('Teacher', self.name, 'teaching a class.')

        def display(self):
                print('name =', self.name)
                print('salary =', self.salary)

class TA(Teacher, Student):
        def __init__(self, name, gpa, salary):
                super().__init__(name, gpa)

        def dowork(self):
                super().doWork()
                print('TA', self.name, 'grading homework.')

        def display(self):
                super().display()

def main():
        ta = TA('Peter', 3.8, 1000)
        ta.dowork()
        ta.display()

main()
```

```
Teacher Peter teaching a class.
TA Peter grading homework.

name = Peter
salary = 3.8
```

```
class Student:
        def __init__(self, name, gpa):
                self.name = name
                self.gpa = gpa

        def dowork(self):
                print('Student', self.name, 'doing homework.')

        def display(self):
                print('name =', self.name)
                print('gpa =', self.gpa)

class Teacher:
        def __init__(self, name, salary, project):
                self.name = name
                self.salary = salary
                self.project = project

        def dowork(self):
                print('Teacher', self.name, 'teaching a class.')

        def display(self):
                print('name =', self.name)
                print('salary =', self.salary)
                print('project =', self.project)

class TA(Teacher, Student):
        def __init__(self, name, gpa, salary, project):
                #super().__init__(name, gpa)  # TypeError: __init__() missing 1
required positional argument: 'project'
                super().__init__(name, salary, project)

        def dowork(self):
                super().dowork()
                print('TA', self.name, 'grading homework.')

        def display(self):
                super().display()


def main():
        ta = TA('Peter', 3.8, 1000, 'MAX5')
        ta.dowork()
        ta.display()
```

Problem of Multiple Inheritance - The left parent class (Teacher) was selected first for __init__(), dowork(), and display().

- The right class (Student) was uninitialized

```
Teacher Peter teaching a class.
TA Peter grading homework.

name = Peter
salary = 1000
project = MAX5
```

```
class Student:
        def __init__(self, name, gpa):
                self.name = name
                self.gpa = gpa

        def dowork(self):
                print('Student', self.name, 'doing homework.')

        def display(self):
                print('name =', self.name)
                print('gpa =', self.gpa)

class Teacher:
        def __init__(self, name, salary, project):
                self.name = name
                self.salary = salary
                self.project = project

        def dowork(self):
                print('Teacher', self.name, 'teaching a class.')

        def display(self):
                print('name =', self.name)
                print('salary =', self.salary)
                print('project =', self.project)

class TA(Teacher, Student):
        def __init__(self, name, gpa, salary, project):
                Teacher.__init__(self, name, salary, project)
                Student.__init__(self, name, gpa)

        def doWork(self):
                super().dowork()
                print('TA', self.name, 'grading homework.')

        def display(self):
                super().display()

def main():
        ta = TA('Peter', 3.8, 1000, 'MAX5')
        ta.dowork()
        ta.display()

main()
```

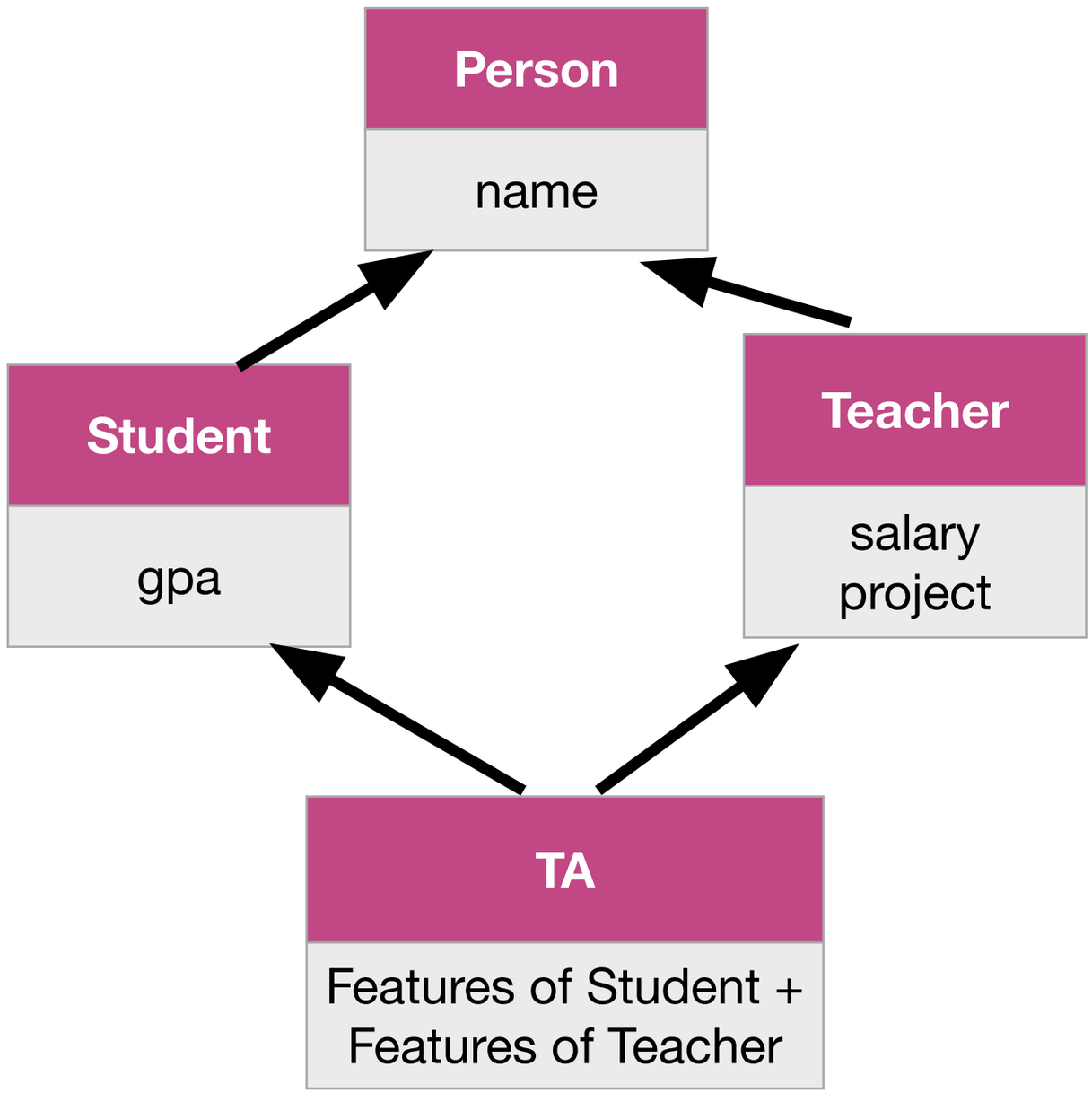
Problem of Multiple Inheritance - To solve the problem using native approach.

```
Teacher Peter teaching a class.
TA Peter grading homework.
name = Peter

salary = 1000
project = MAX5
```

The diamond problems

- Since Student and Teacher's _init_() require different number of arguments, you will have to call each __init__() explicitly using native approach, i.e. Student.__init__() and Teacher.__init__()
- Possibility of calling superclass's methods multiple times using native approach



Example of Diamond Problems - native approach

```
class Person:
        def __init__(self, name):
                self.name = name

        def dowork(self):
                print('Person:', self.name, 'doing nothing.')

        def display(self):
                print('Person: name =', self.name)

class Student(Person):
        def __init__(self, name, gpa):
                Person.__init__(self, name)
                self.gpa = gpa

        def dowork(self):
                Person.dowork(self);
                print('Student', self.name, 'doing homework.')

        def display(self):
                Person.display(self)
                print('Student: gpa =', self.gpa)

class Teacher(Person):
        def __init__(self, name, salary, project):
                Person.__init__(self, name)
                self.salary = salary
                self.project = project

        def dowork(self):
                Person.dowork(self);
                print('Teacher', self.name, 'teaching a class.')

        def display(self):
                Person.display(self)
                print('Teacher: salary =', self.salary)
                print('Teacher: project =', self.project)

class TA(Teacher, Student):
        def __init__(self, name, gpa, salary, project):
                Teacher.__init__(self, name, salary, project)
                Student.__init__(self, name, gpa)

        def dowork(self):
                Teacher.dowork(self)
                Student.dowork(self)
                print('TA:', self.name, 'grading homework.')

        def display(self):
                Teacher.display(self)
                Student.display(self)


def main():
        ta = TA('Peter', 3.8, 1000, 'MAX5')
        ta.dowork()
        ta.display()
        print()
        ta.name = "Lily"
        print()
        ta.display()
        print()

main()
```

```
Person: Peter doing nothing.
Teacher Peter teaching a class.

Person: Peter doing nothing.
Student Peter doing homework.

TA: Peter grading homework.

Person: name = Peter
Teacher: salary = 1000
Teacher: project = MAX5

Person: name = Peter
Student: gpa = 3.8


Person: name = Lily
Teacher: salary = 1000
Teacher: project = MAX5

Person: name = Lily
Student: gpa = 3.8
```

Example of Diamond Problems - super() approach

```
class Person:
        def __init__(self, name):
                self.name = name

        def dowork(self):
                print('Person:', self.name, 'doing nothing.')

        def display(self):
                print('Person: name =', self.name)

class Student(Person):
        def __init__(self, name, gpa):
                Person.__init__(self, name)
                self.gpa = gpa

        def dowork(self):
                super().dowork();
                print('Student', self.name, 'doing homework.')

        def display(self):
                super().display()
                print('Student: gpa =', self.gpa)

class Teacher(Person):
        def __init__(self, name, salary, project):
                Person.__init__(self, name)
                self.salary = salary
                self.project = project

        def dowork(self):
                super().dowork();
                print('Teacher', self.name, 'teaching a class.')

        def display(self):
                super().display()
                print('Teacher: salary =', self.salary)
                print('Teacher: project =', self.project)

class TA(Teacher, Student):
        def __init__(self, name, gpa, salary, project):
                Teacher.__init__(self, name, salary, project)
                Student.__init__(self, name, gpa)

        def dowork(self):
                super().dowork()
                print('TA:', self.name, 'grading homework.')

        def display(self):
                super().display()

def main():
        ta = TA('Peter', 3.8, 1000, 'MAX5')
        ta.dowork()
        ta.display()
        print()
        ta.name = "Lily"
        print()
        ta.display()
        print()

main()
```

```
Person: Peter doing nothing.
Student Peter doing homework.
Teacher Peter teaching a class.
TA: Peter grading homework.

Person: name = Peter
Student: gpa = 3.8
Teacher: salary = 1000
Teacher: project = MAX5


Person: name = Lily
Student: gpa = 3.8
Teacher: salary = 1000
Teacher: project = MAX5
```

Abstract base class

- An abstract class can be considered as a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class.
- A class which contains one or more abstract methods is called an abstract class.
- An abstract method is a method that has a declaration but does not have an implementation.
- Using an abstract class, we can provide a common interface for different implementations of a software component.
- An abstract class is a class that can be inherited by other classes but they you can’t use to create an object.

Why and How Abstract base class?

- Using an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses. This allows us to provide interfaces (like contracts) to a software contractor to implement interfaces.
- By default, Python does not provide abstract classes. Python comes with a module which provides the base for defining Abstract Base classes(ABC) and that module name is ABC. ABC works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword @abstractmethod.

Example of Abstract Base Class - Can’t be instantiated

```
from abc import ABC, abstractmethod, abstractproperty

class Parent(ABC):
    @abstractmethod
    def dowork(self):
        pass

    @abstractmethod
    def make_money(self):
        pass

class Child(Parent):
    def dowork(self):
        print("Learn computer programming!")

class GrandChild(Child):
    def make_money(self):
        print("Stock trading!")

def main():
    # TypeError: Can't instantiate abstract class Parent with abstract methods doWork, makeMoney
    # p = Parent()

    # TypeError: Can't instantiate abstract class Child with abstract methods makeMoney
    # c = Child()

    g = GrandChild()
    g.dowork()
    g.make_money()


main()
```

```
Learn computer programming!
Stock trading!
```

Example of Abstract Class Appication

```
from abc import ABC, abstractmethod, abstractproperty

class Person(ABC):
        def __init__(self, name):
                self.__name = name

        @abstractproperty
        def gpa(self):
                return "parent class"

        @abstractmethod
        def dowork(self):
                pass

        @property
        def name(self):
                return self.__name

        def display(self):
                print('Person:', self.name)

class Student(Person):
        def __init__(self, name, gpa):
                super().__init__(name)
                self.__gpa = gpa

        @property
        def gpa(self):
                return self.__gpa

        def dowork(self):
                print("Doing homework!")

        def display(self):
                super().display()
                print('Student:', self.gpa)

def main():t
        a = Student('Peter', 3.8)
        a.dowork()
        a.display()

main()
```

```
Doing homework!
Person: Peter
Student: 3.8
```

Dictionaries

- In a dictionary, each item consists of a key/value pair where each value in the dictionary is indexed by a unique key.
- The key can be any immutable data type, but it's usually a string.
- The value for a key can be a simple data type such as a number or a string. Or, it can be a complex data type such as a list or another dictionary.
- For details, please refer to the following URLs:
  - https://docs.python.org/3/tutorial/datastructures.html
  - https://www.w3schools.com/python/python_ref_dictionary.asp

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Dictionaries

- You can use the del keyword or the pop() and clear() methods to delete items from a dictionary.
- example,

```
courses = {“CS204”: “C Programming”,
                “CS360”: “C++ Programming”,
                 “CS480”: “Java Programming”}
id = “CS360”
if id in courses:
    course = courses[id]
    del courses[id]
```

Looping

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```