```
from abc import ABC, abstractmethod
import re


class Equation(ABC):
    degree: int
    type: str

    def __init__(self, *args):
        if (self.degree + 1) != len(args):
            raise TypeError(
                f"'Equation' object takes {self.degree + 1} positional
arguments but {len(args)} were given"
            )
        if any(not isinstance(arg, (int, float)) for arg in args):
            raise TypeError("Coefficients must be of type 'int' or
'float'")
        if args[0] == 0:
            raise ValueError("Highest degree coefficient must be
different from zero")
        self.coefficients = {(len(args) - n - 1): arg for n, arg in
enumerate(args)}

    def __init_subclass__(cls):
        if not hasattr(cls, "degree"):
            raise AttributeError(
                f"Cannot create '{cls.__name__}' class: missing
required attribute 'degree'"
            )
        if not hasattr(cls, "type"):
            raise AttributeError(
                f"Cannot create '{cls.__name__}' class: missing
required attribute 'type'"
            )

    def __str__(self):
        terms = []
        for n, coefficient in self.coefficients.items():
            if not coefficient:
                continue
            if n == 0:
                terms.append(f'{coefficient:+}')
            elif n == 1:
                terms.append(f'{coefficient:+}x')
            else:
                terms.append(f"{coefficient:+}x**{n}")
        equation_string = ' '.join(terms) + ' = 0'
        return re.sub(r"(?<!\d)1(?=x)", "",
equation_string.strip("+"))
```

```
    @abstractmethod
    def solve(self):
        pass

    @abstractmethod
    def analyze(self):
        pass


class LinearEquation(Equation):
    degree = 1
    type = 'Linear Equation'

    def solve(self):
        a, b = self.coefficients.values()
        x = -b / a
        return [x]

    def analyze(self):
        slope, intercept = self.coefficients.values()
        return {'slope': slope, 'intercept': intercept}


class QuadraticEquation(Equation):
    degree = 2
    type = 'Quadratic Equation'

    def __init__(self, *args):
        super().__init__(*args)
        a, b, c = self.coefficients.values()
        self.delta = b**2 - 4 * a * c

    def solve(self):
        if self.delta < 0:
            return []
        a, b, _ = self.coefficients.values()
        x1 = (-b + (self.delta) ** 0.5) / (2 * a)
        x2 = (-b - (self.delta) ** 0.5) / (2 * a)
        if self.delta == 0:
            return [x1]

        return [x1, x2]

    def analyze(self):
        a, b, c = self.coefficients.values()
        x = -b / (2 * a)
        y = a * x**2 + b * x + c
        if a > 0:
            concavity = 'upwards'
            min_max = 'min'
```

```
        else:
            concavity = 'downwards'
            min_max = 'max'
        return {'x': x, 'y': y, 'min_max': min_max, 'concavity':
concavity}


def solver(equation):
    if not isinstance(equation, Equation):
        raise TypeError("Argument must be an Equation object")

    output_string = f'\n{equation.type:-^24}'
    output_string += f'\n\n{equation!s:^24}\n\n'
    output_string += f'{"Solutions":-^24}\n\n'
    results = equation.solve()
    match results:
        case []:
            result_list = ['No real roots']
        case [x]:
            result_list = [f'x = {x:+.3f}']
        case [x1, x2]:
            result_list = [f'x1 = {x1:+.3f}', f'x2 = {x2:+.3f}']
    for result in result_list:
        output_string += f'{result:^24}\n'
    output_string += f'\n{"Details":-^24}\n\n'
    details = equation.analyze()
    match details:
        case {'slope': slope, 'intercept': intercept}:
            details_list = [f'slope = {slope:>16.3f}', f'y-intercept =
{intercept:>10.3f}']
        case {'x': x, 'y': y, 'min_max': min_max, 'concavity':
concavity}:
            coord = f'({x:.3f}, {y:.3f})'
            details_list = [f'concavity = {concavity}', f'{min_max} =
{coord}']
    for detail in details_list:
        output_string += f'{detail}\n'
    return output_string
lin_eq = LinearEquation(2, 3)
quadr_eq = QuadraticEquation(1, 2, 1)
print(solver(quadr_eq))
```