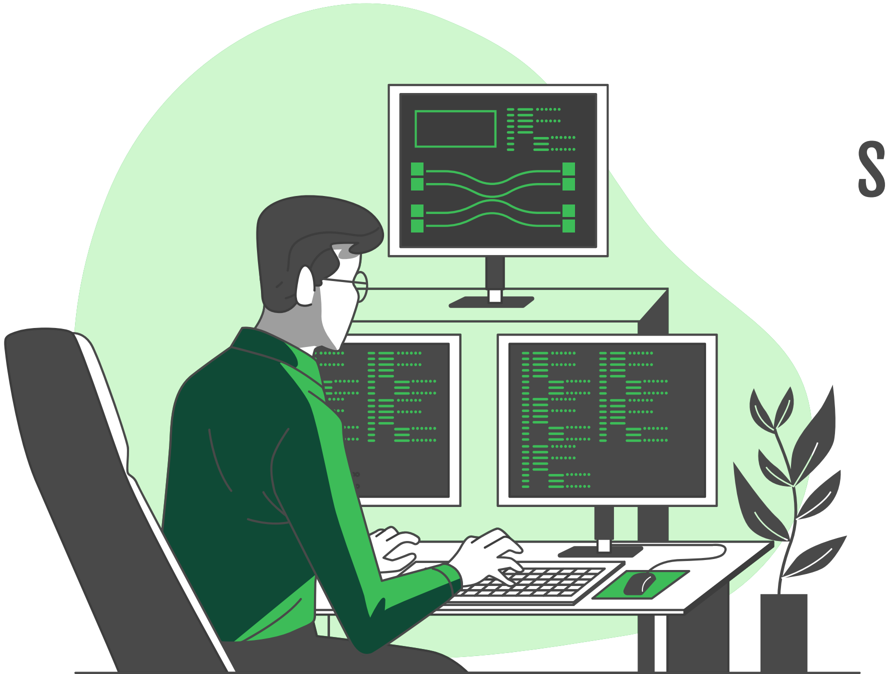
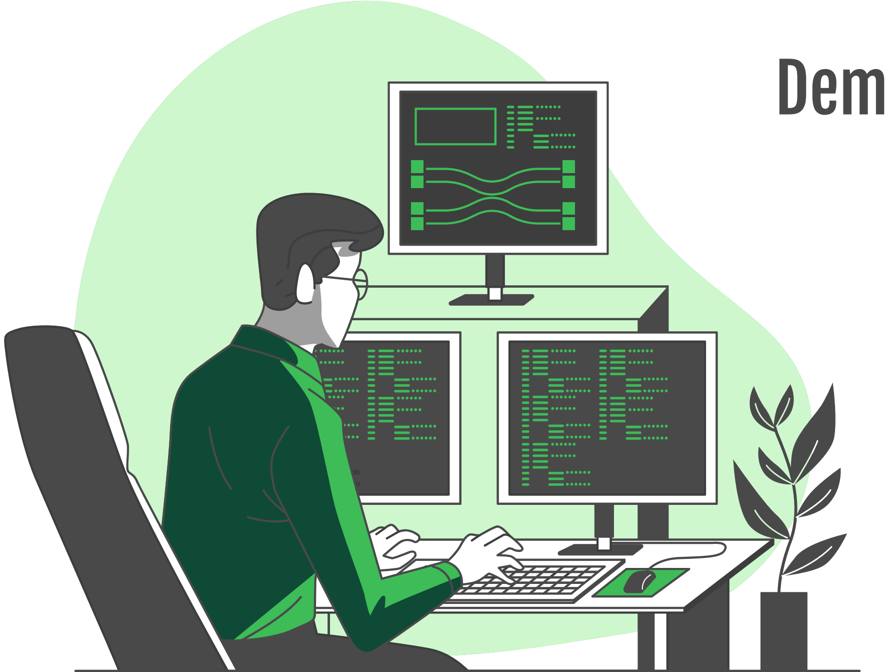


Desmitificando el Principio de Sustitución de Liskov con nuestro lenguaje más querido: Python

Juan David Alzate Cardona



Demystifying the Liskov Substitution Principle with our most loved programming language: Python

Juan David Alzate Cardona

About me

- Software Engineer at Hourly.
- Physics lover.
- Python lover.
- Teaching lover.
- Dogs lover.

Hourly

PYCON COLOMBIA 2023

Table of Contents

01 Introduction

02 Understanding Subtyping

03 The Liskov Substitution Principle

04 Guidelines for Applying LSP

05 Using Mypy to check the LSP

06 Benefits and Consequences of Violating LSP

07 Summary and Conclusion

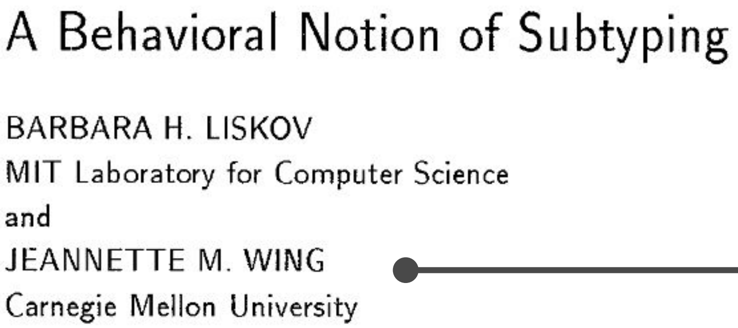
01

Introduction

SOLID

SOLID

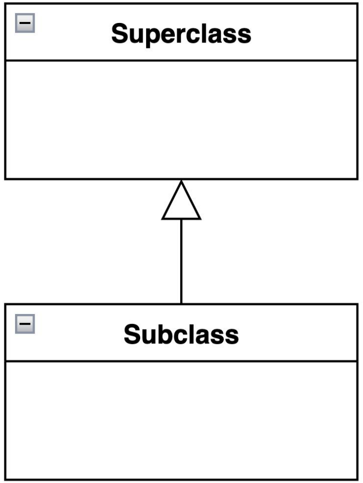
Liskov Substitution Principle



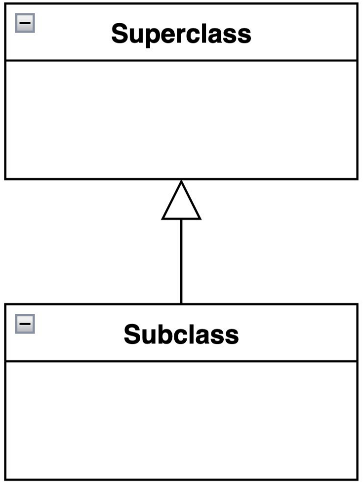
A Behavioral Notion of Subtyping

BARBARA H. LISKOV
MIT Laboratory for Computer Science
and
JEANNETTE M. WING
Carnegie Mellon University

The Liskov Substitution Principle (LSP) states that objects of a **superclass** should be replaceable with objects of its **subclasses** without altering the **correctness** of a program.

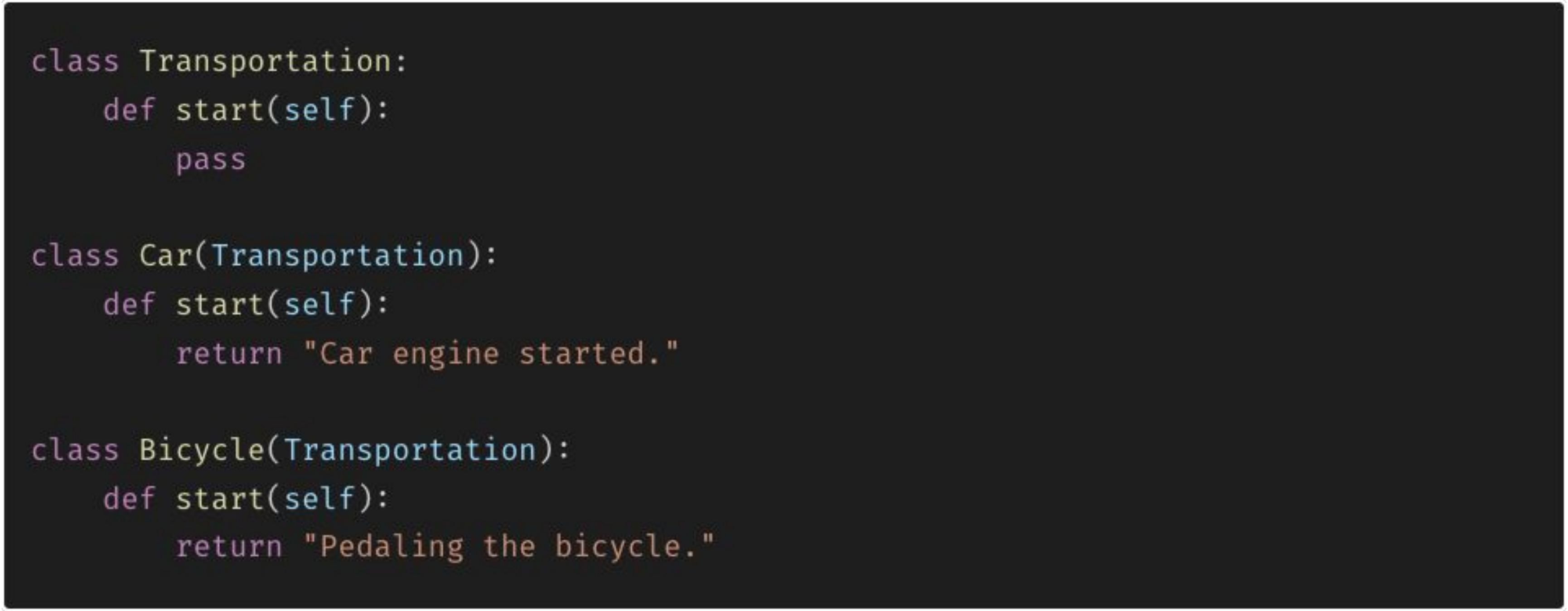


The Liskov Substitution Principle (LSP) states that objects of a **superclass** should be replaceable with objects of its **subclasses** without altering the **correctness** of a program.



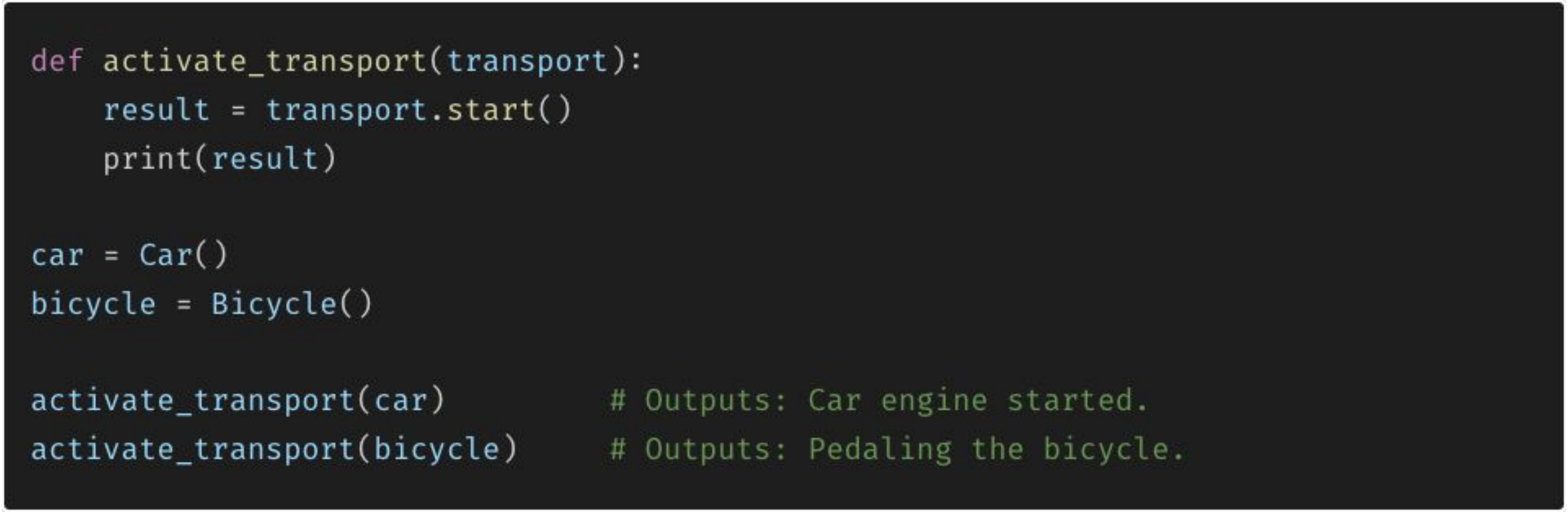
- Modularity
- Code reuse
- Extensibility
- Easy substitution

Transportation example



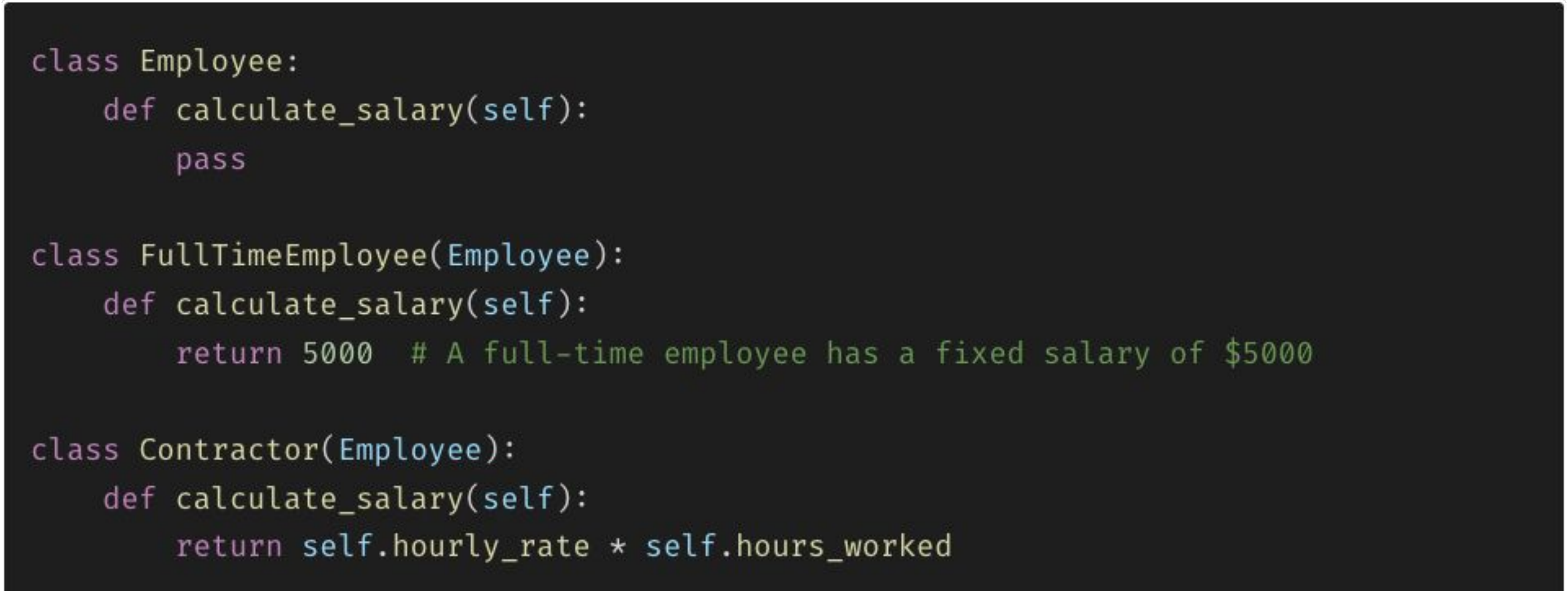
```python
class Transportation:
    def start(self):
        pass

class Car(Transportation):
    def start(self):
        return "Car engine started."

class Bicycle(Transportation):
    def start(self):
        return "Pedaling the bicycle."
```

Transportation example



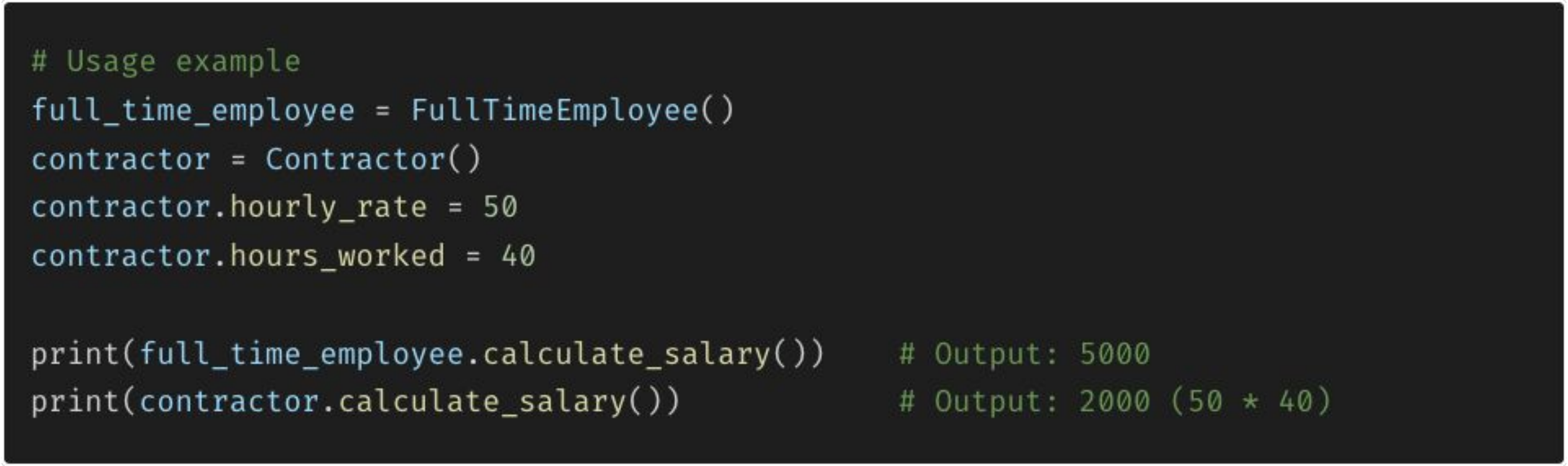
```python
def activate_transport(transport):
    result = transport.start()
    print(result)

car = Car()
bicycle = Bicycle()

activate_transport(car)         # Outputs: Car engine started.
activate_transport(bicycle)     # Outputs: Pedaling the bicycle.
```

Employee example



```python
class Employee:
    def calculate_salary(self):
        pass

class FullTimeEmployee(Employee):
    def calculate_salary(self):
        return 5000  # A full-time employee has a fixed salary of $5000

class Contractor(Employee):
    def calculate_salary(self):
        return self.hourly_rate * self.hours_worked
```

Employee example



```python
# Usage example
full_time_employee = FullTimeEmployee()
contractor = Contractor()
contractor.hourly_rate = 50
contractor.hours_worked = 40

print(full_time_employee.calculate_salary())   # Output: 5000
print(contractor.calculate_salary())           # Output: 2000 (50 * 40)
```

Why Python?

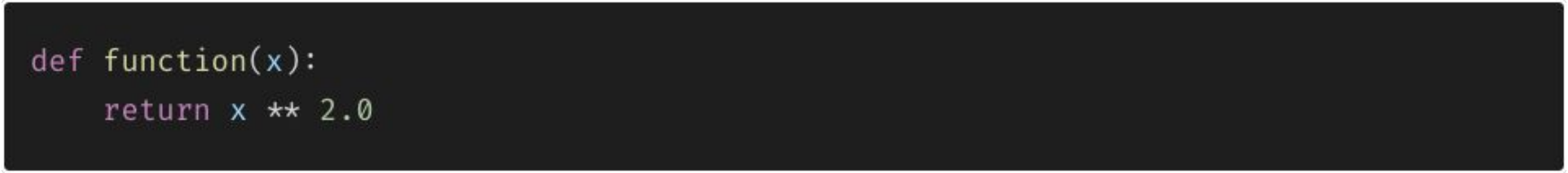
Dynamic typing

PEP 484 – Type Hints

Why Python?

Dynamic typing

PEP 484 – Type Hints

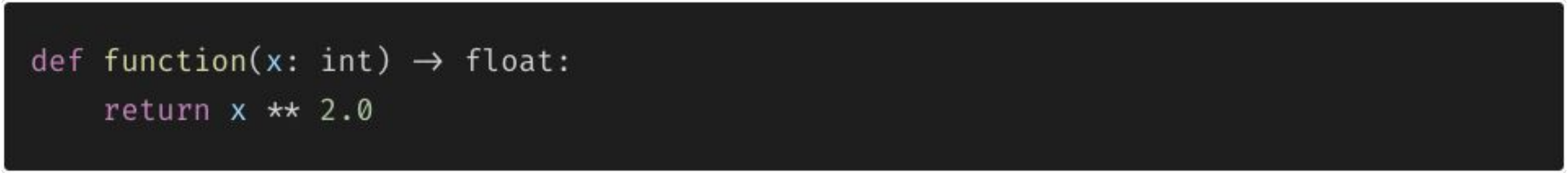


```
def function(x):
    return x ** 2.0
```

Why Python?

Dynamic typing

PEP 484 – Type Hints

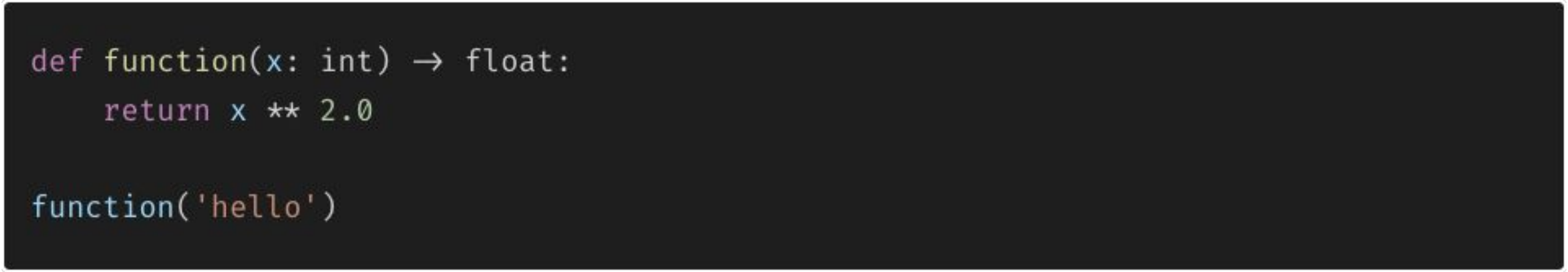


```python
def function(x: int) -> float:
    return x ** 2.0
```

Why Python?

Dynamic typing

PEP 484 – Type Hints

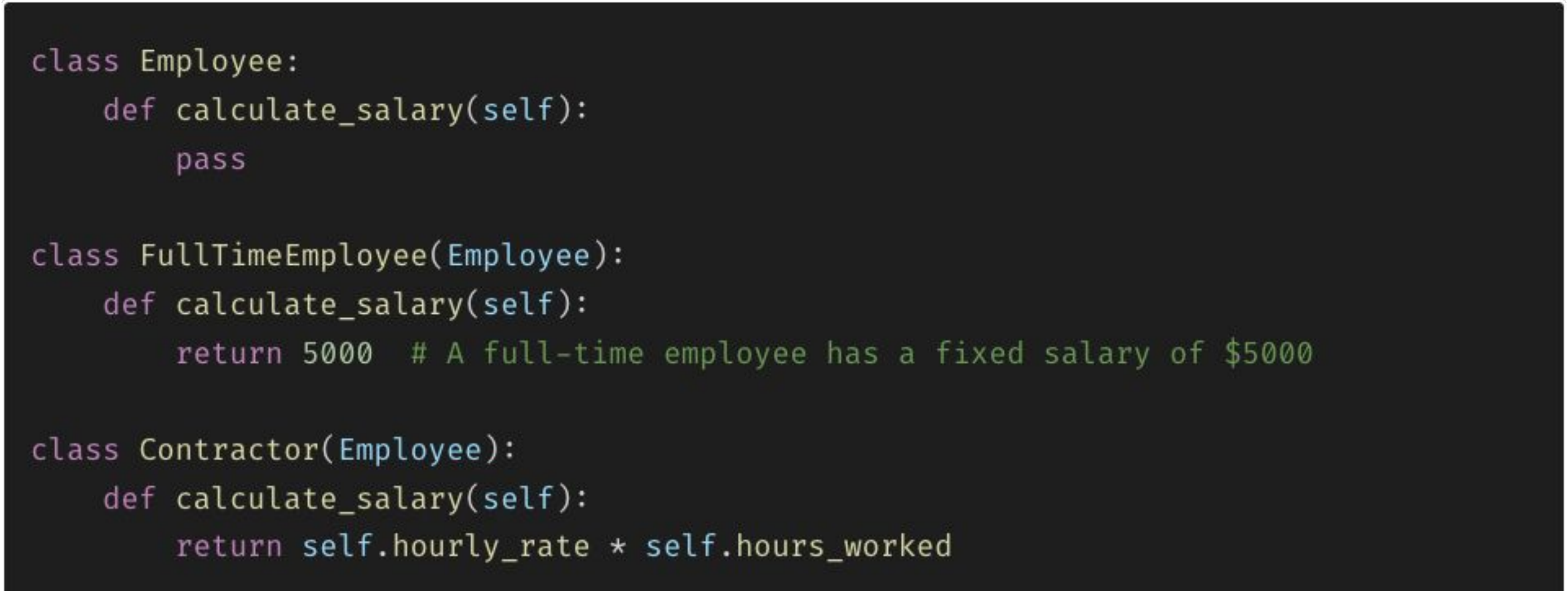


```
def function(x: int) → float:
    return x ** 2.0

function('hello')
```

02

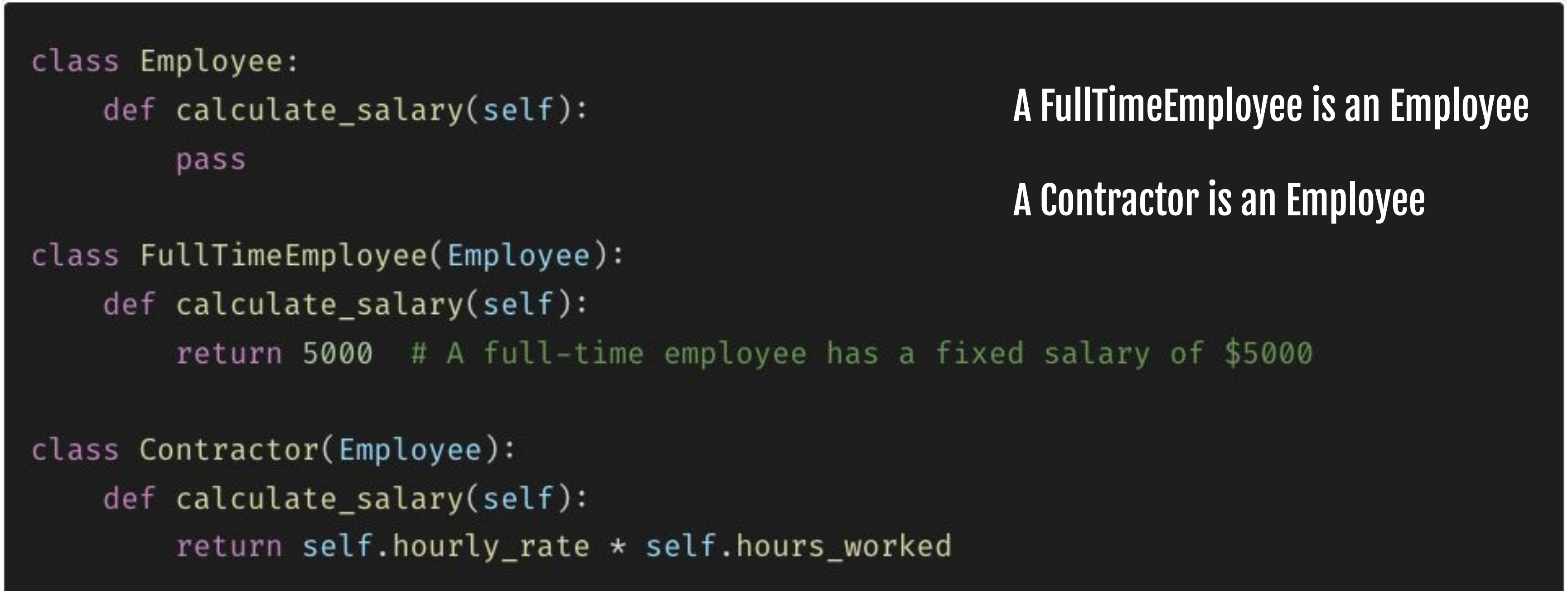
Understanding Subtyping

Employee example



```python
class Employee:
    def calculate_salary(self):
        pass

class FullTimeEmployee(Employee):
    def calculate_salary(self):
        return 5000  # A full-time employee has a fixed salary of $5000

class Contractor(Employee):
    def calculate_salary(self):
        return self.hourly_rate * self.hours_worked
```

Employee example

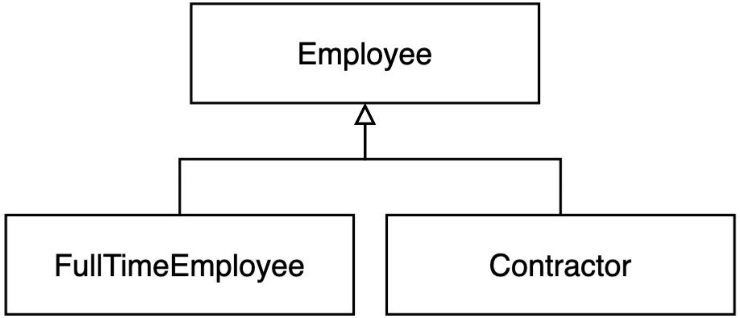


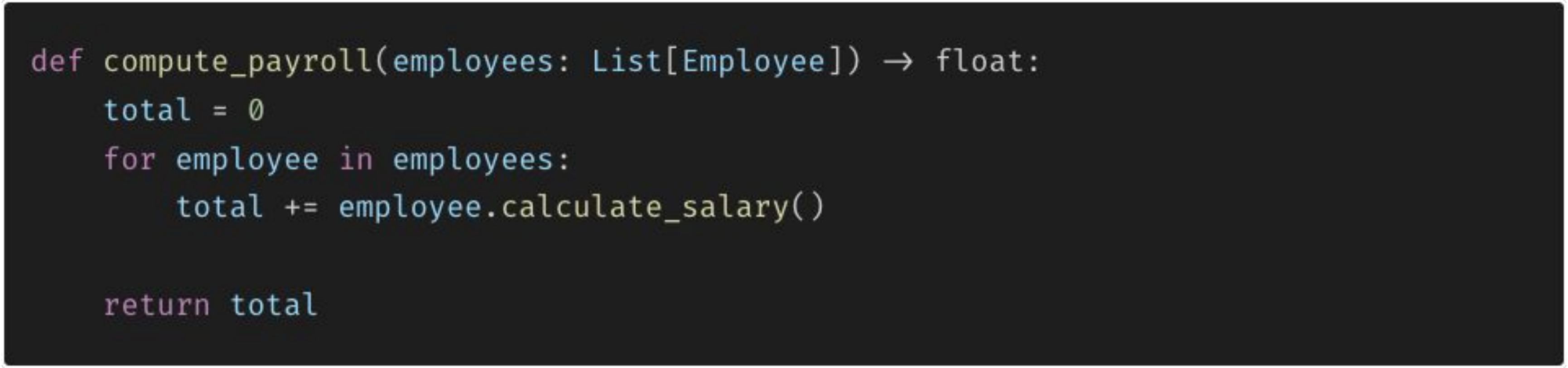
```
class Employee:
    def calculate_salary(self):
        pass

class FullTimeEmployee(Employee):
    def calculate_salary(self):
        return 5000  # A full-time employee has a fixed salary of $5000

class Contractor(Employee):
    def calculate_salary(self):
        return self.hourly_rate * self.hours_worked
```

A FullTimeEmployee is an Employee

A Contractor is an Employee

Substitutability

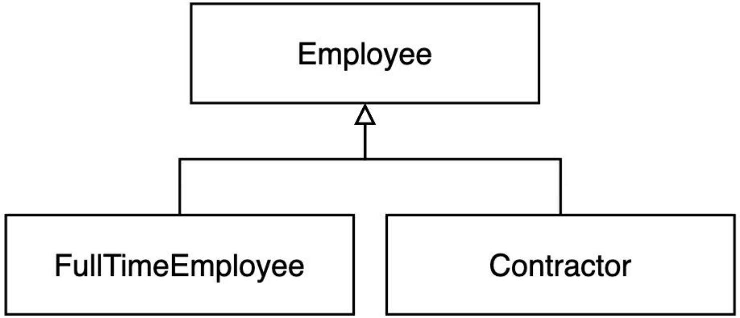


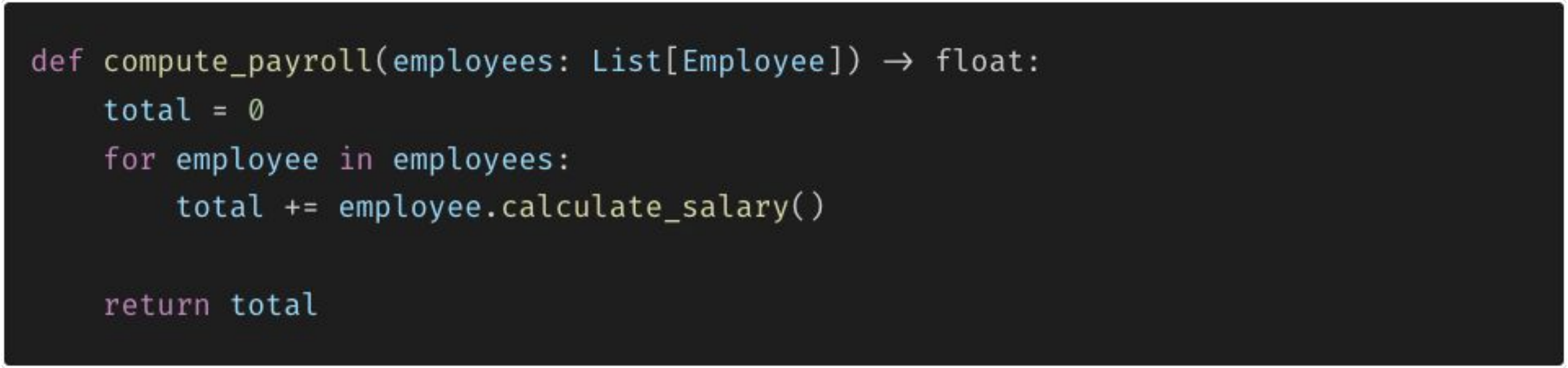


```python
def compute_payroll(employees: List[Employee]) -> float:
    total = 0
    for employee in employees:
        total += employee.calculate_salary()

    return total
```

Substitutability

Is a → Can substitute to





```python
def compute_payroll(employees: List[Employee]) -> float:
    total = 0
    for employee in employees:
        total += employee.calculate_salary()

    return total
```

Subtyping → Substitutability

Code reusability

Less code to write

Abstraction

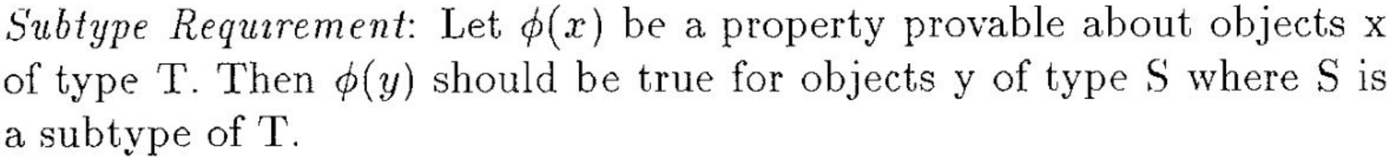
Encapsulate data and behavior

Polymorphism

Depends on specific implementations

03

The Liskov Substitution Principle



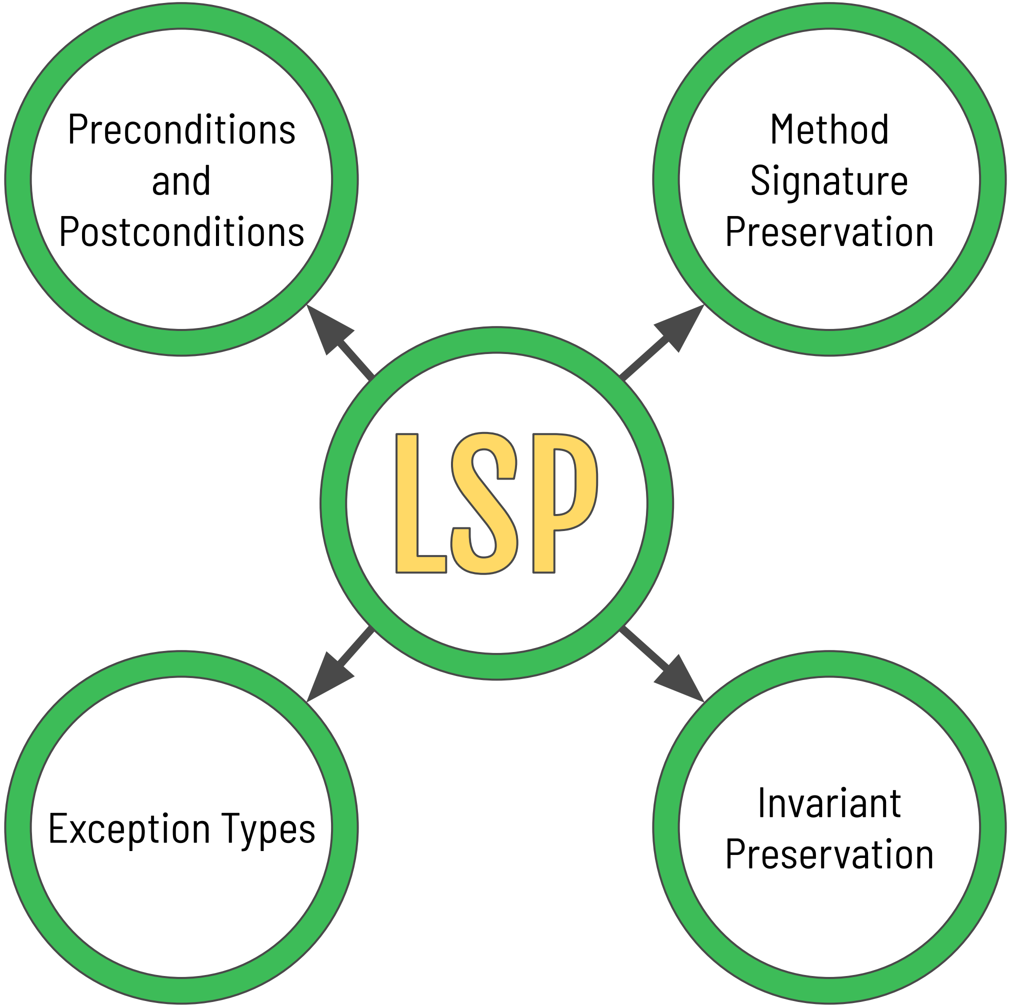
Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

If a program is using **an object of a superclass** and the program expects certain properties or behaviors to hold true for that object, then the program should still work correctly if **an object of any of its subclasses is used instead**.

The subclass object should be able to substitute the superclass object without affecting the **correctness** of the program.

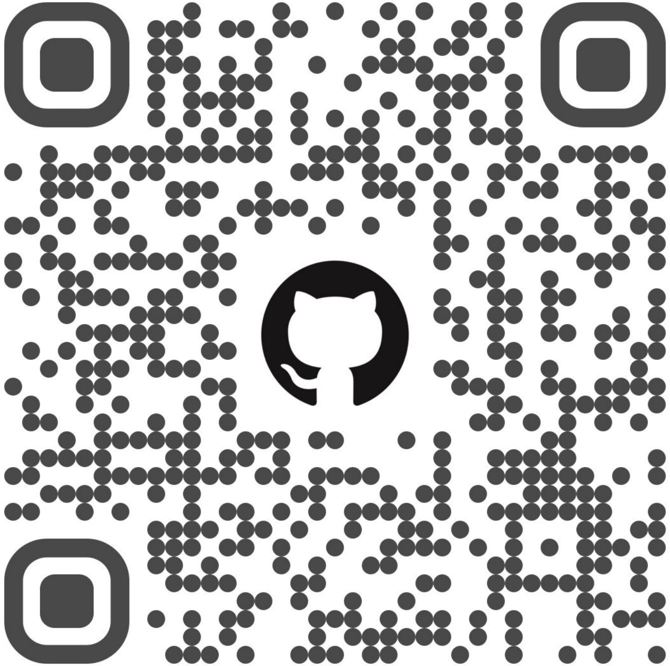
04

Guidelines for Applying LSP



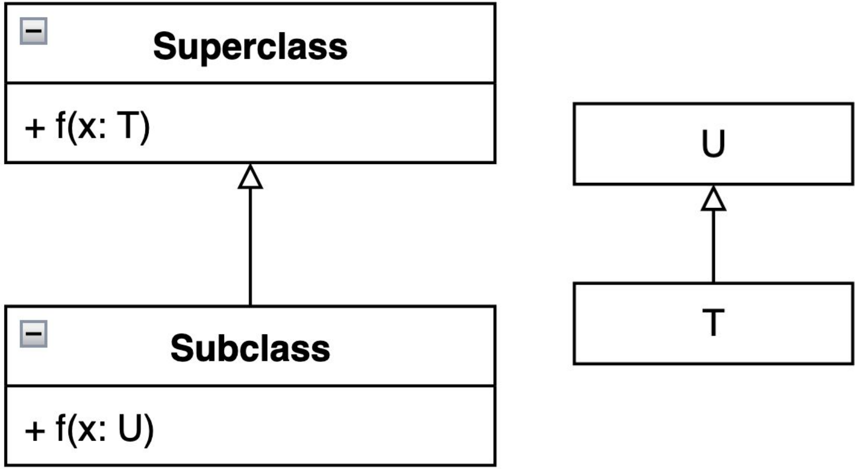
Preconditions and Postconditions
Method Signature Preservation
LSP
Exception Types
Invariant Preservation

Code



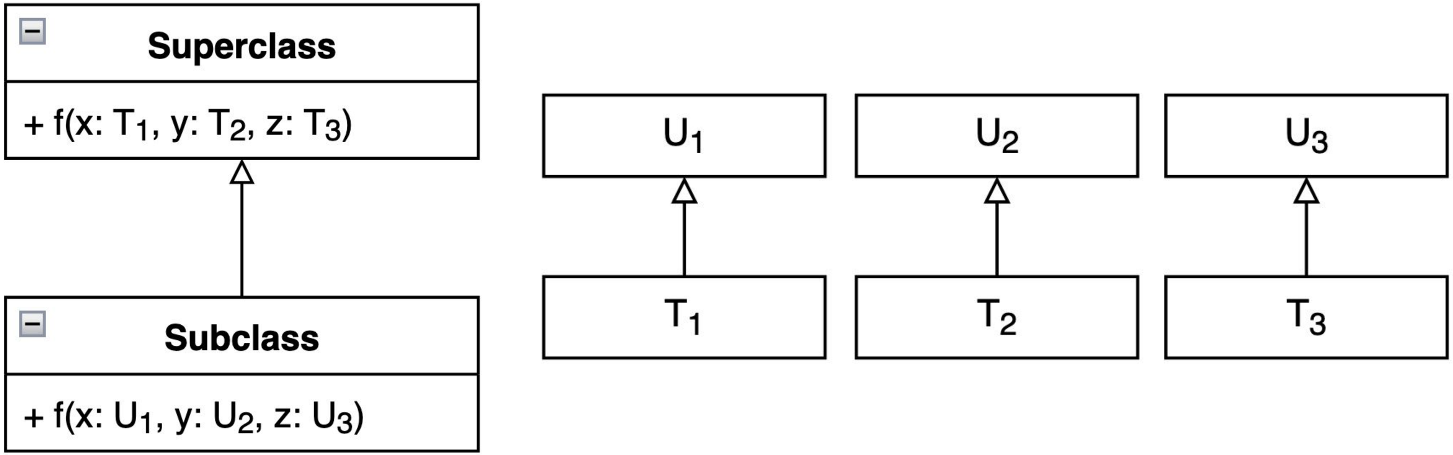
Check 1

In a **subclass**, the parameter types of a method should either **match** the parameter types in the corresponding method of the superclass or be **more general** (or abstract) than the parameter types in the superclass.



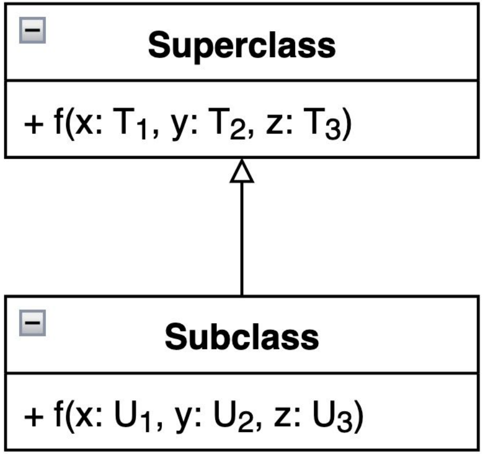
Check 1

In a **subclass**, the parameter types of a method should either **match** the parameter types in the corresponding method of the superclass or be **more general** (or abstract) than the parameter types in the superclass.


Superclass
+ f(x: T_1, y: T_2, z: T_3)
Subclass
+ f(x: U_1, y: U_2, z: U_3)
U_1
U_2
U_3
T_1
T_2
T_3

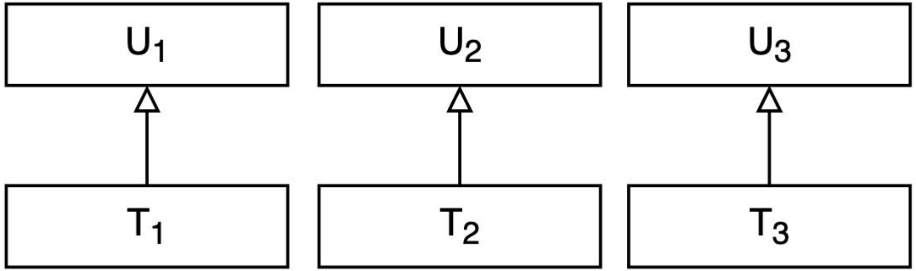
Check 1

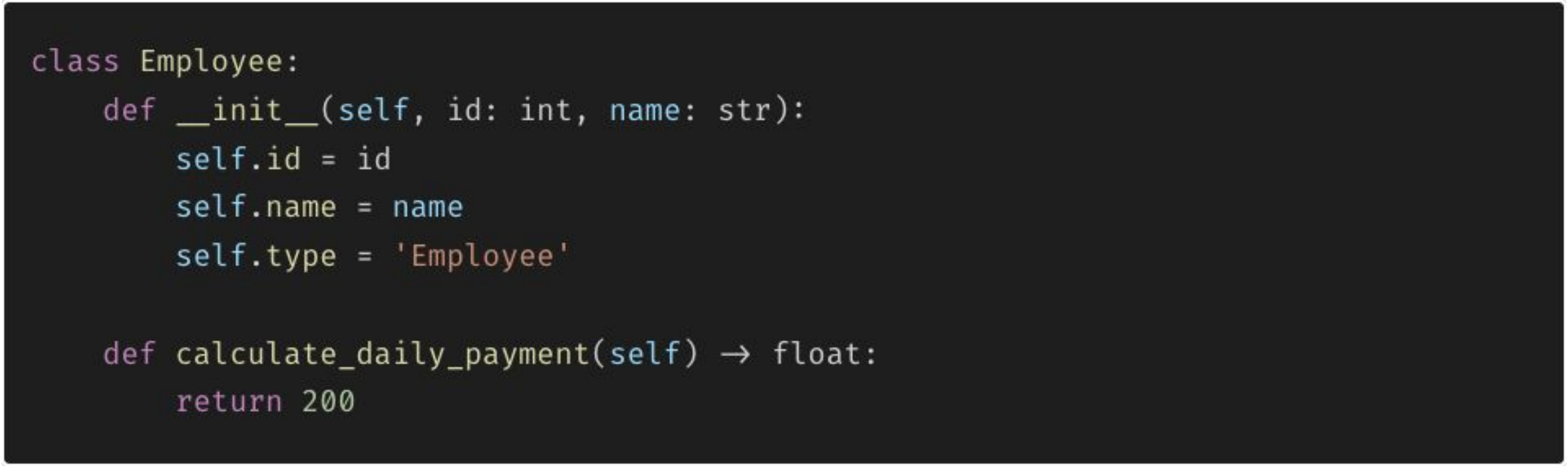
In a **subclass**, the parameter types of a method should either **match** the parameter types in the corresponding method of the superclass or be **more general** (or abstract) than the parameter types in the superclass.



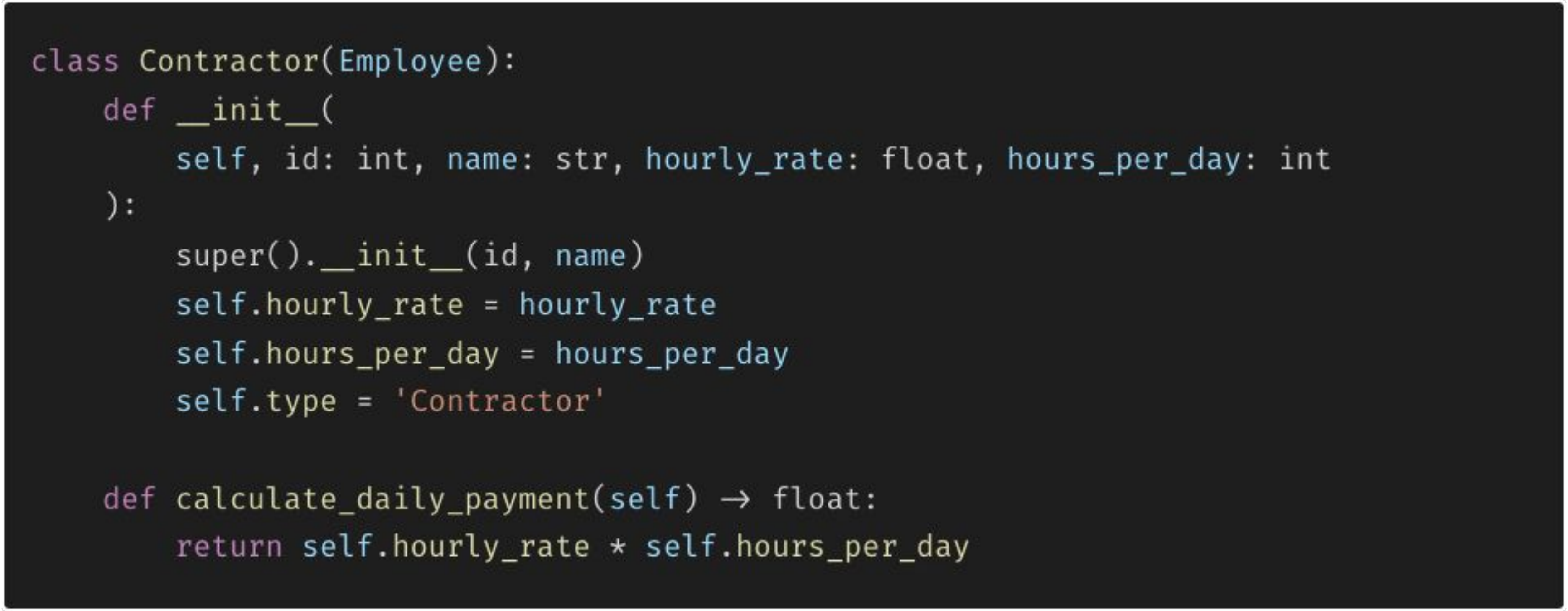
Superclass
+ f(x: T_1, y: T_2, z: T_3)
Subclass
+ f(x: U_1, y: U_2, z: U_3)

Contravariance

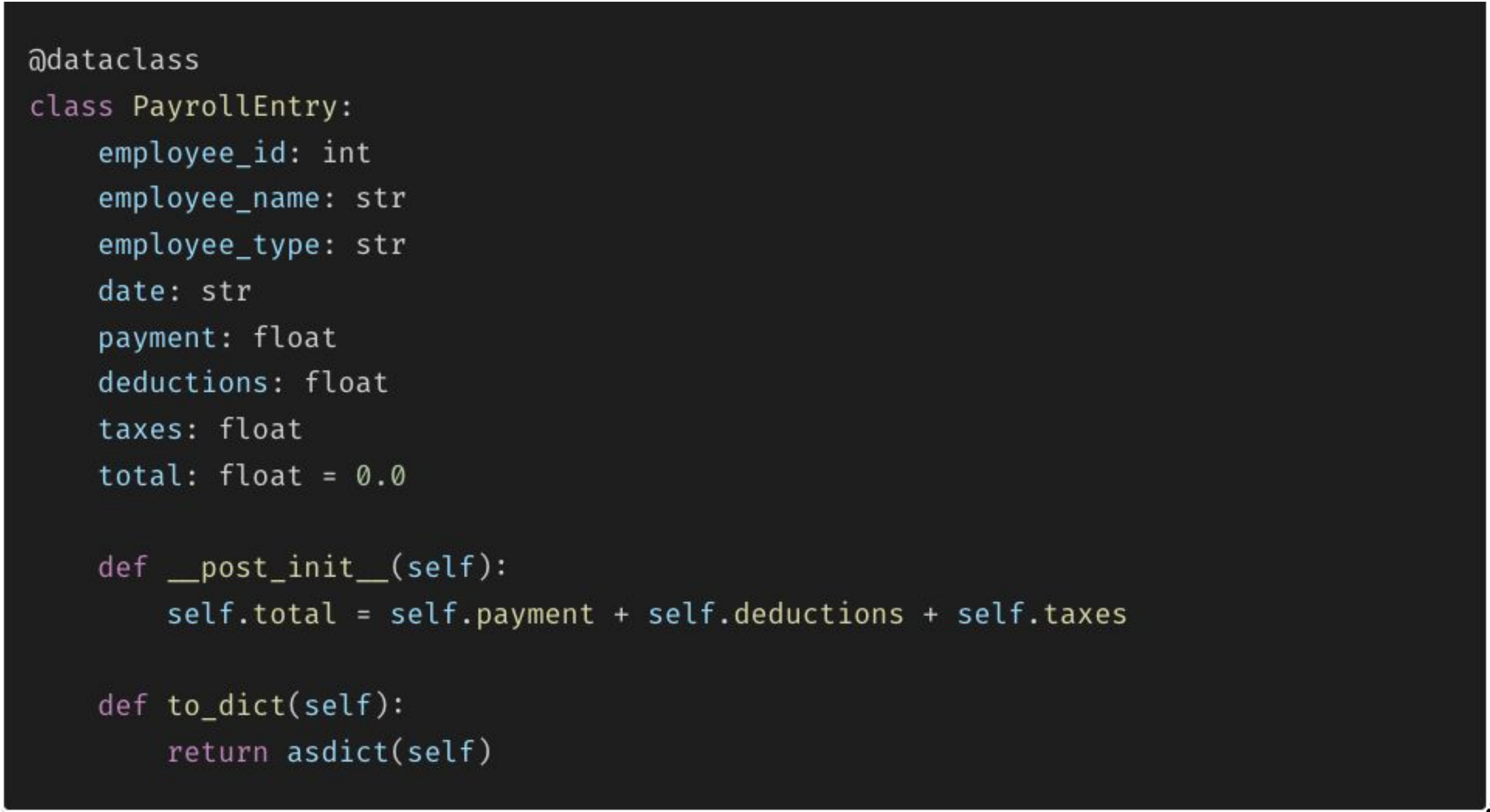




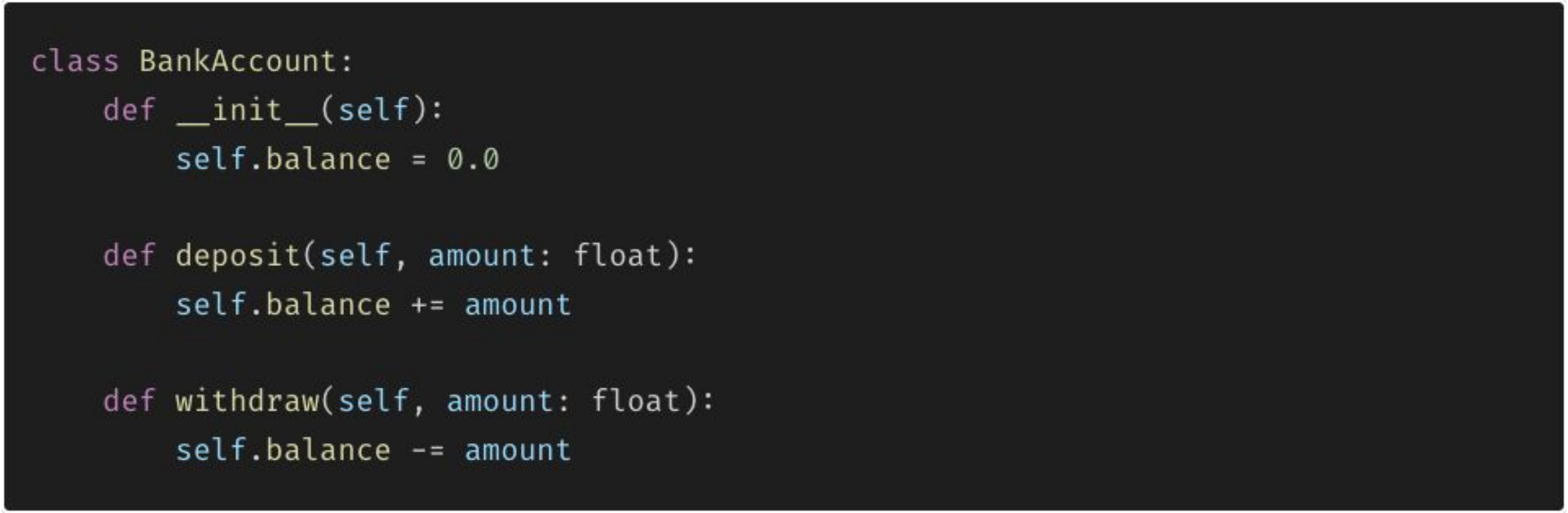
```python
class Employee:
    def __init__(self, id: int, name: str):
        self.id = id
        self.name = name
        self.type = 'Employee'

    def calculate_daily_payment(self) -> float:
        return 200
```



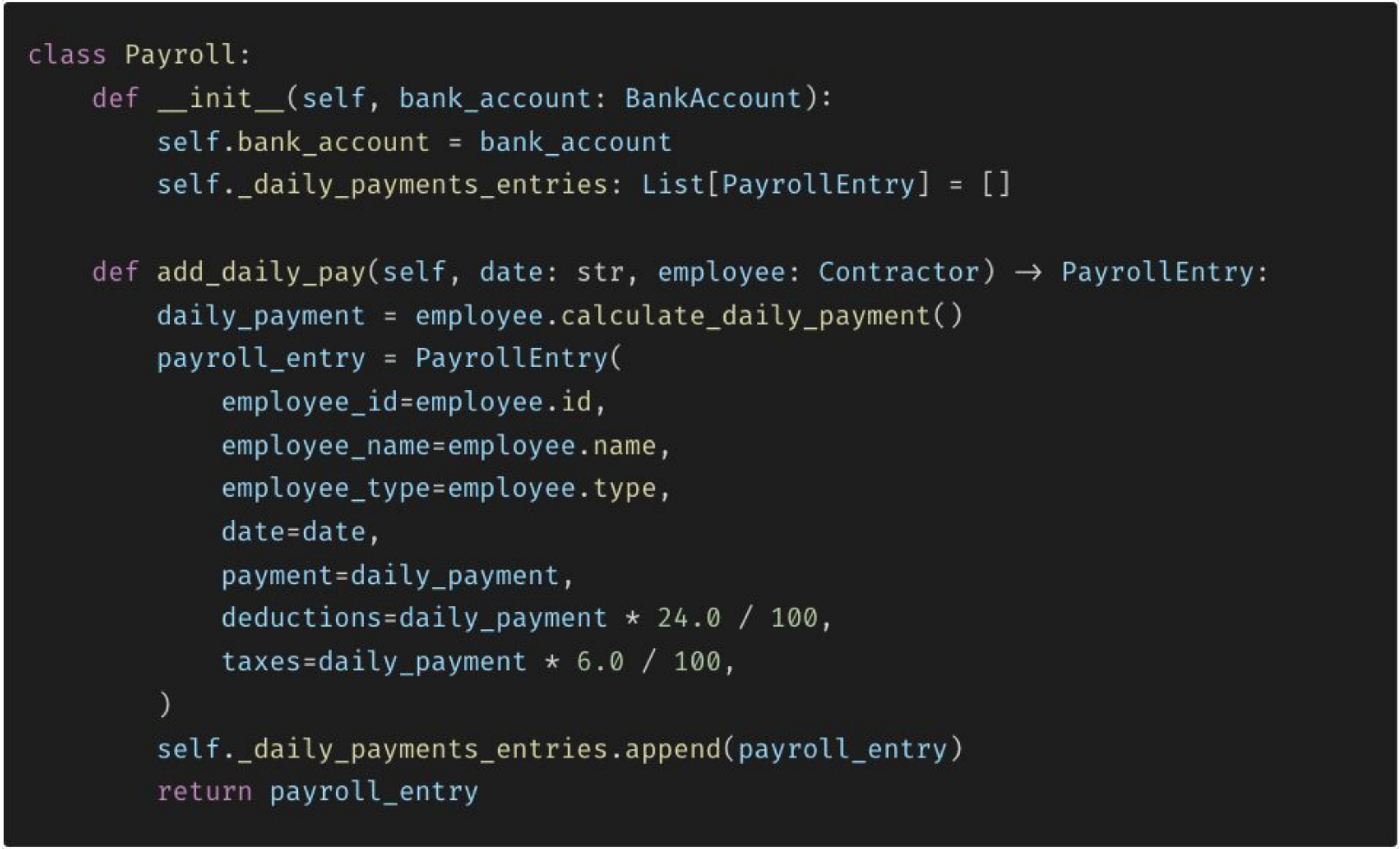
```python
class Contractor(Employee):
    def __init__(
        self, id: int, name: str, hourly_rate: float, hours_per_day: int
    ):
        super().__init__(id, name)
        self.hourly_rate = hourly_rate
        self.hours_per_day = hours_per_day
        self.type = 'Contractor'

    def calculate_daily_payment(self) -> float:
        return self.hourly_rate * self.hours_per_day
```



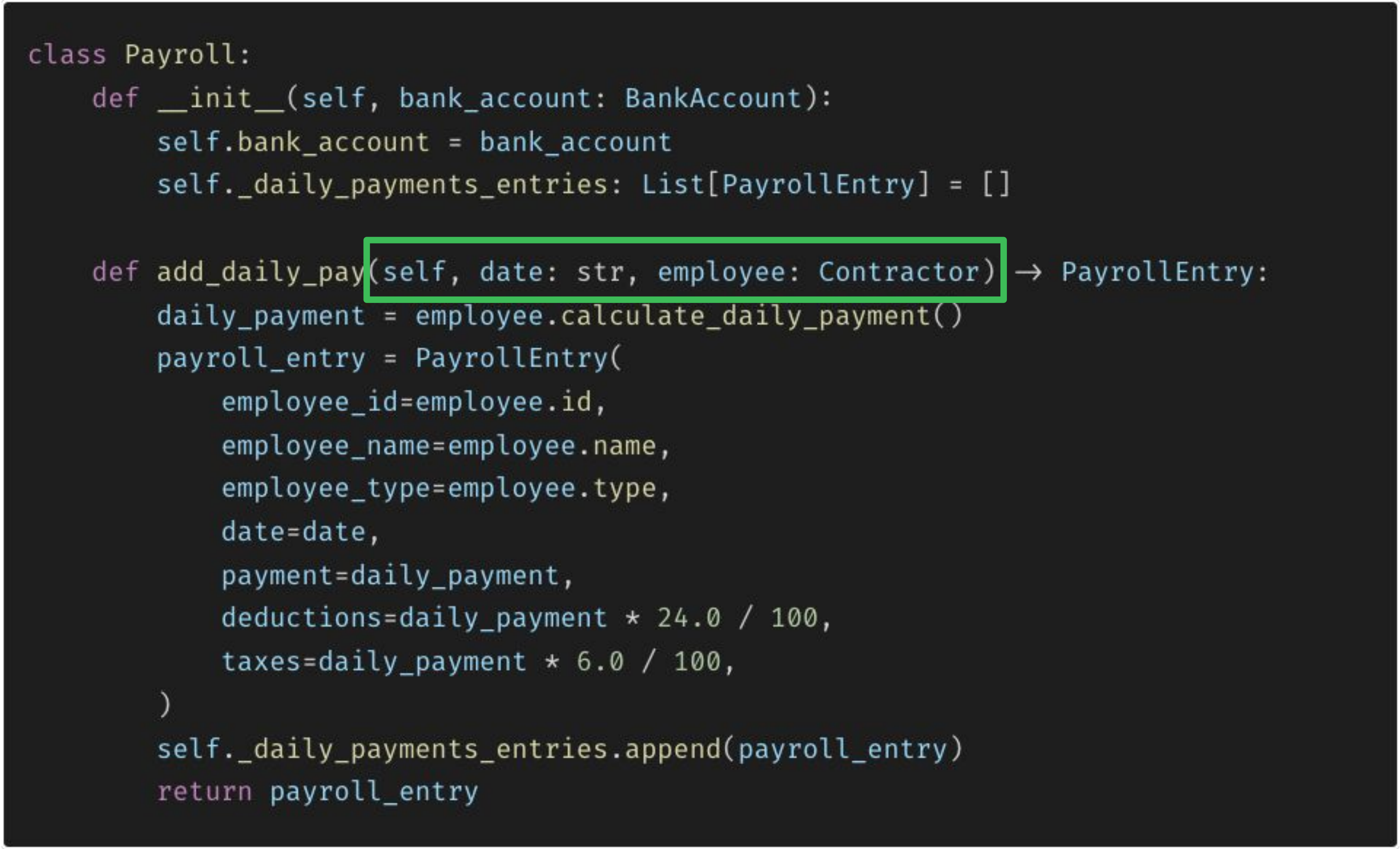
```python
@dataclass
class PayrollEntry:
    employee_id: int
    employee_name: str
    employee_type: str
    date: str
    payment: float
    deductions: float
    taxes: float
    total: float = 0.0

    def __post_init__(self):
        self.total = self.payment + self.deductions + self.taxes

    def to_dict(self):
        return asdict(self)
```



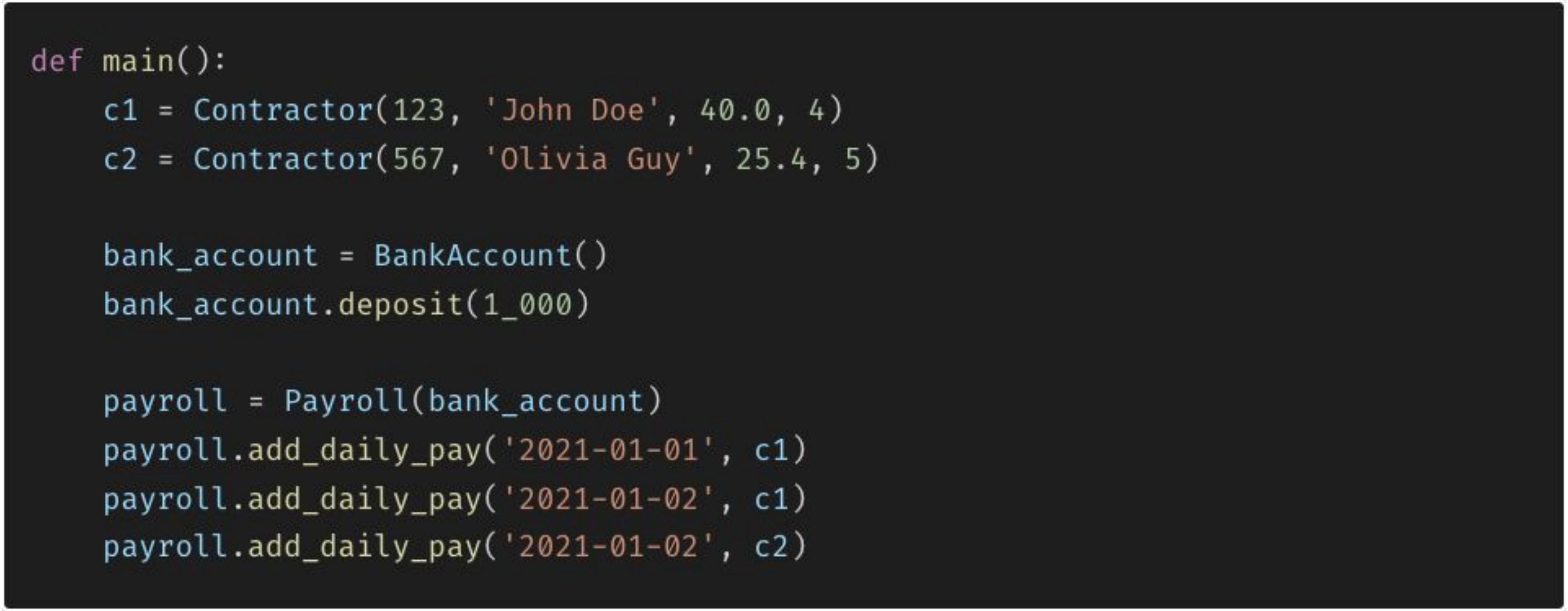
```python
class BankAccount:
    def __init__(self):
        self.balance = 0.0

    def deposit(self, amount: float):
        self.balance += amount

    def withdraw(self, amount: float):
        self.balance -= amount
```



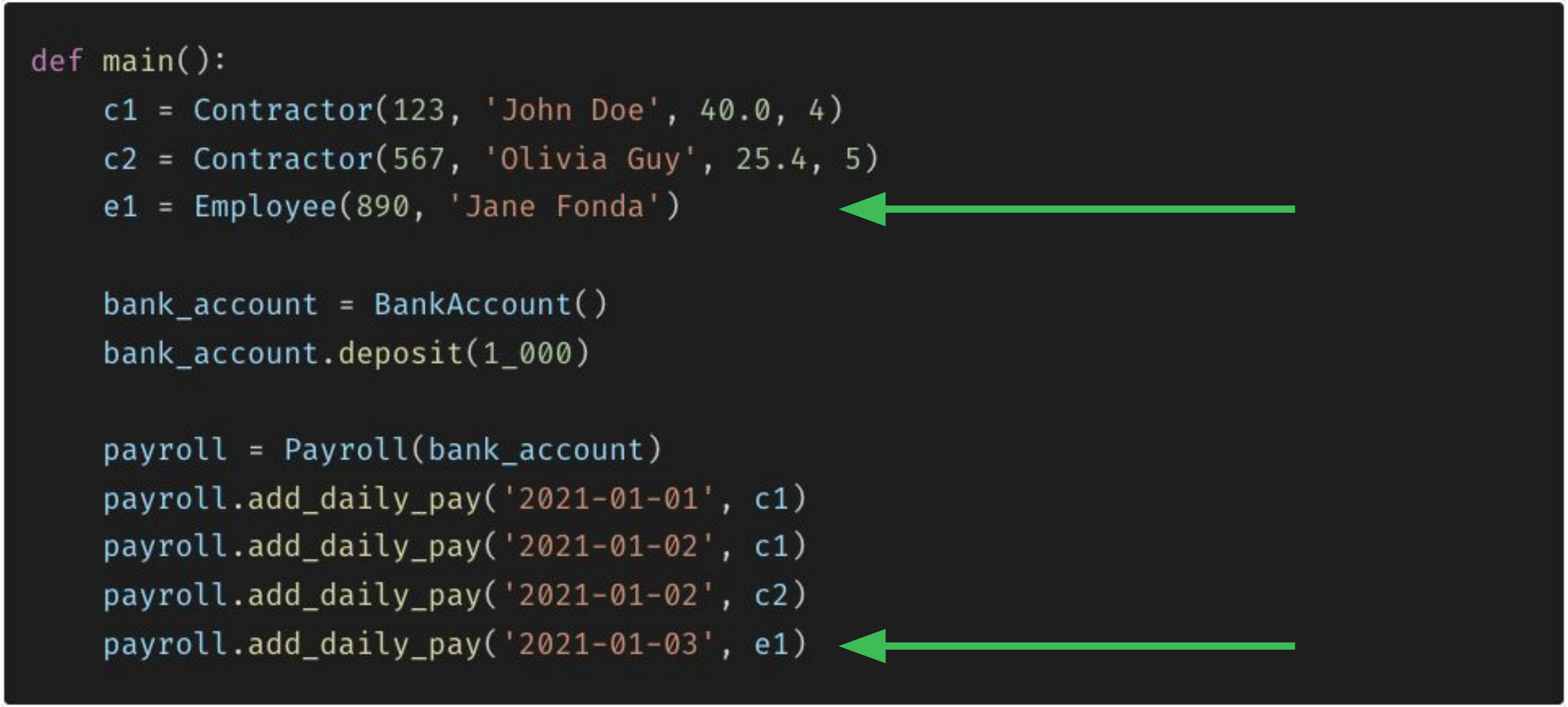
```python
class Payroll:
    def __init__(self, bank_account: BankAccount):
        self.bank_account = bank_account
        self._daily_payments_entries: List[PayrollEntry] = []

    def add_daily_pay(self, date: str, employee: Contractor) -> PayrollEntry:
        daily_payment = employee.calculate_daily_payment()
        payroll_entry = PayrollEntry(
            employee_id=employee.id,
            employee_name=employee.name,
            employee_type=employee.type,
            date=date,
            payment=daily_payment,
            deductions=daily_payment * 24.0 / 100,
            taxes=daily_payment * 6.0 / 100,
        )
        self._daily_payments_entries.append(payroll_entry)
        return payroll_entry
```



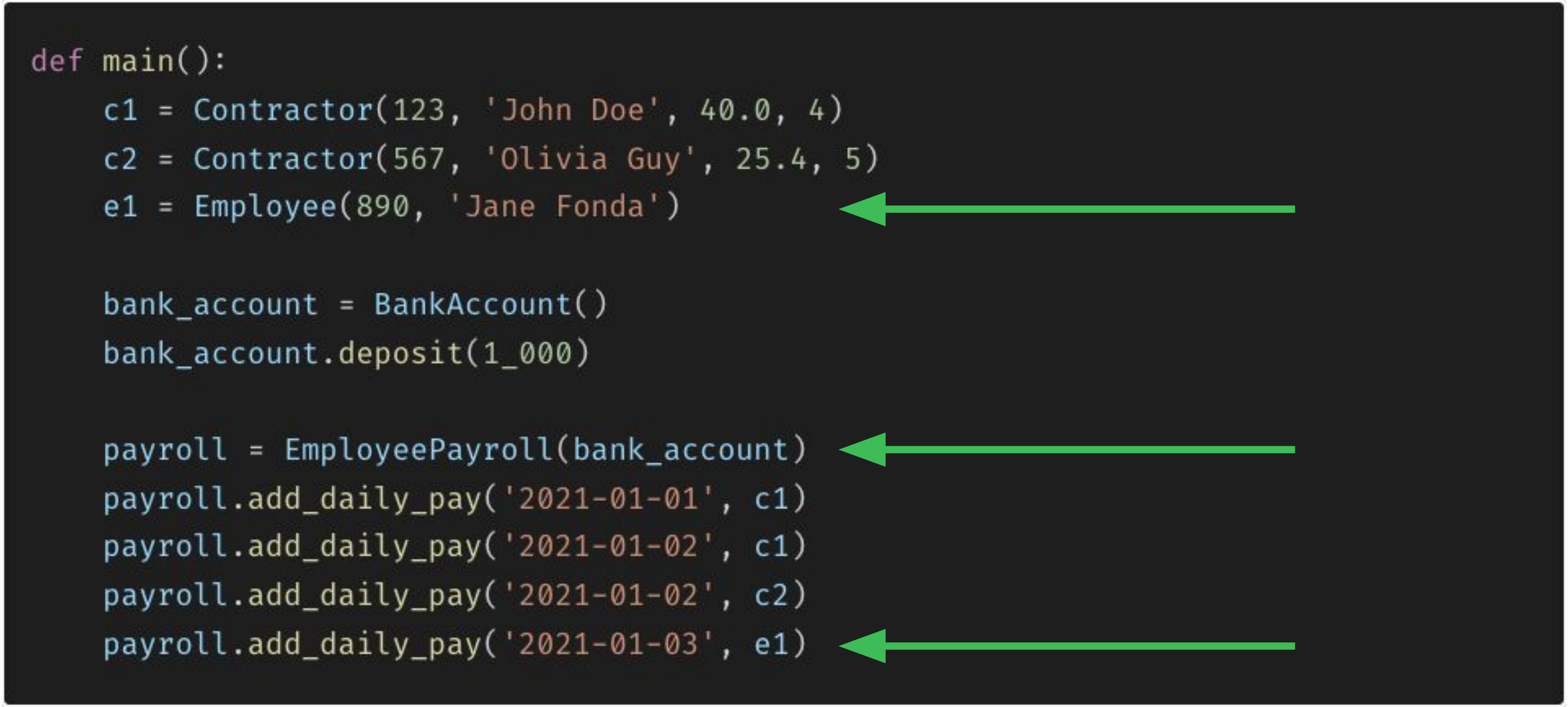
```python
class Payroll:
    def __init__(self, bank_account: BankAccount):
        self.bank_account = bank_account
        self._daily_payments_entries: List[PayrollEntry] = []

    def add_daily_pay(self, date: str, employee: Contractor) -> PayrollEntry:
        daily_payment = employee.calculate_daily_payment()
        payroll_entry = PayrollEntry(
            employee_id=employee.id,
            employee_name=employee.name,
            employee_type=employee.type,
            date=date,
            payment=daily_payment,
            deductions=daily_payment * 24.0 / 100,
            taxes=daily_payment * 6.0 / 100,
        )
        self._daily_payments_entries.append(payroll_entry)
        return payroll_entry
```



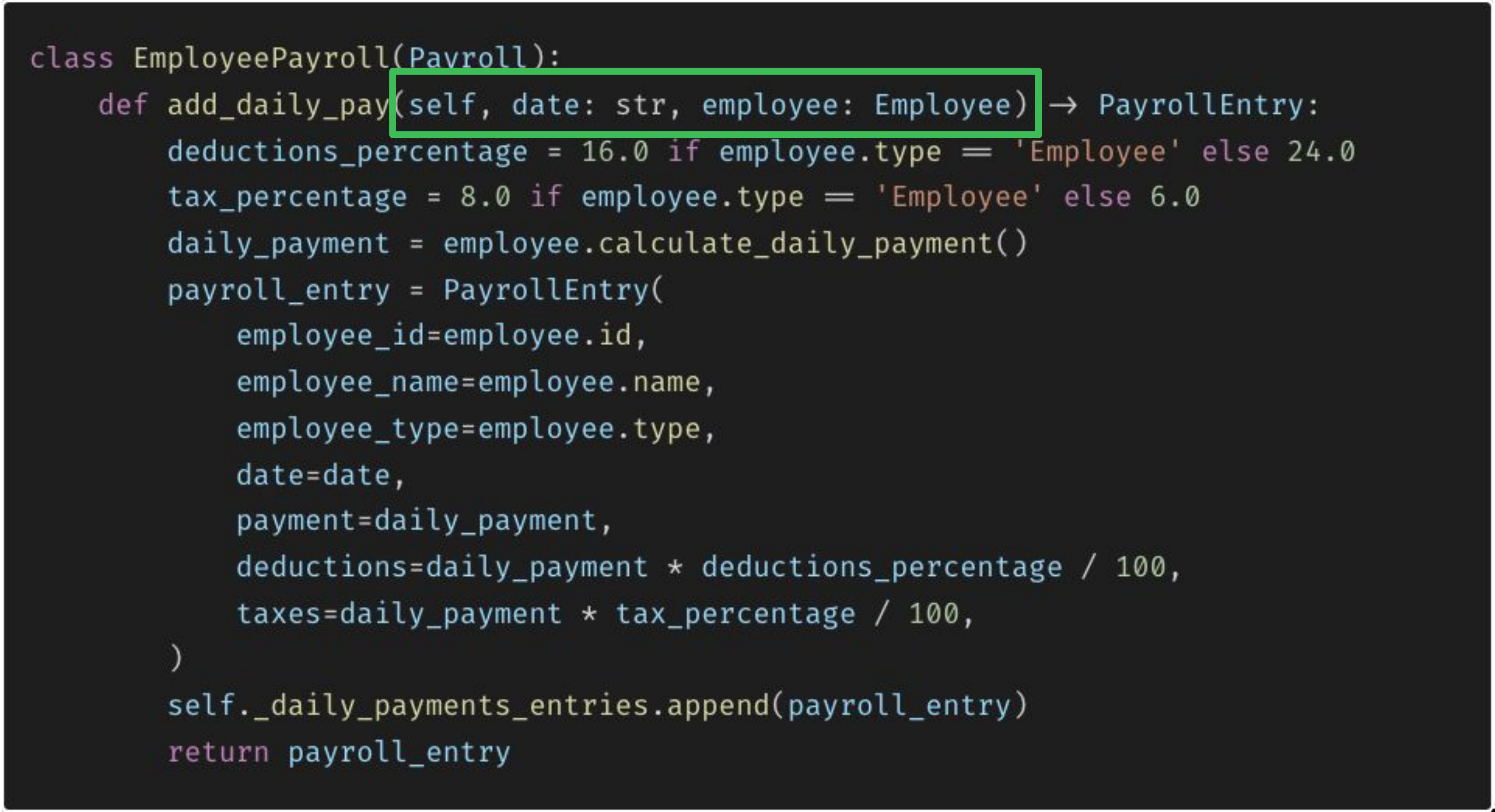
```python
def main():
    c1 = Contractor(123, 'John Doe', 40.0, 4)
    c2 = Contractor(567, 'Olivia Guy', 25.4, 5)

    bank_account = BankAccount()
    bank_account.deposit(1_000)

    payroll = Payroll(bank_account)
    payroll.add_daily_pay('2021-01-01', c1)
    payroll.add_daily_pay('2021-01-02', c1)
    payroll.add_daily_pay('2021-01-02', c2)
```



```
def main():
    c1 = Contractor(123, 'John Doe', 40.0, 4)
    c2 = Contractor(567, 'Olivia Guy', 25.4, 5)
    e1 = Employee(890, 'Jane Fonda')

    bank_account = BankAccount()
    bank_account.deposit(1_000)

    payroll = Payroll(bank_account)
    payroll.add_daily_pay('2021-01-01', c1)
    payroll.add_daily_pay('2021-01-02', c1)
    payroll.add_daily_pay('2021-01-02', c2)
    payroll.add_daily_pay('2021-01-03', e1)
```



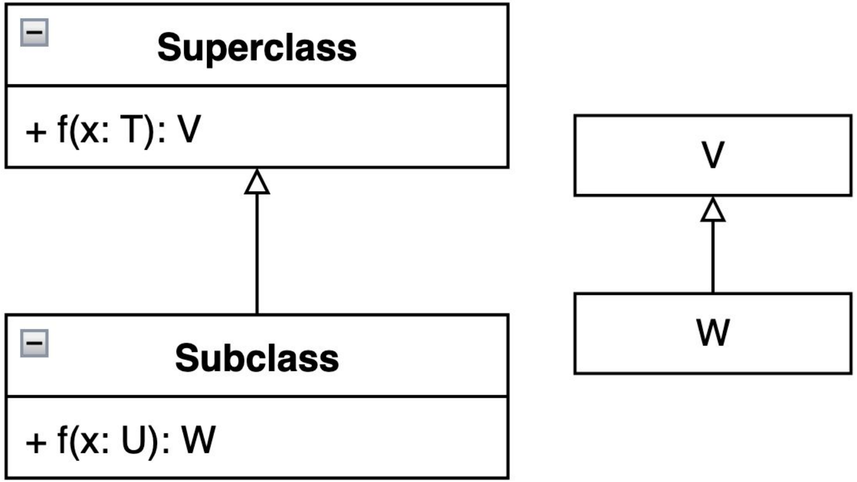
```python
def main():
    c1 = Contractor(123, 'John Doe', 40.0, 4)
    c2 = Contractor(567, 'Olivia Guy', 25.4, 5)
    e1 = Employee(890, 'Jane Fonda')

    bank_account = BankAccount()
    bank_account.deposit(1_000)

    payroll = EmployeePayroll(bank_account)
    payroll.add_daily_pay('2021-01-01', c1)
    payroll.add_daily_pay('2021-01-02', c1)
    payroll.add_daily_pay('2021-01-02', c2)
    payroll.add_daily_pay('2021-01-03', e1)
```



```python
class EmployeePayroll(Payroll):
    def add_daily_pay(self, date: str, employee: Employee) -> PayrollEntry:
        deductions_percentage = 16.0 if employee.type == 'Employee' else 24.0
        tax_percentage = 8.0 if employee.type == 'Employee' else 6.0
        daily_payment = employee.calculate_daily_payment()
        payroll_entry = PayrollEntry(
            employee_id=employee.id,
            employee_name=employee.name,
            employee_type=employee.type,
            date=date,
            payment=daily_payment,
            deductions=daily_payment * deductions_percentage / 100,
            taxes=daily_payment * tax_percentage / 100,
        )
        self._daily_payments_entries.append(payroll_entry)
        return payroll_entry
```

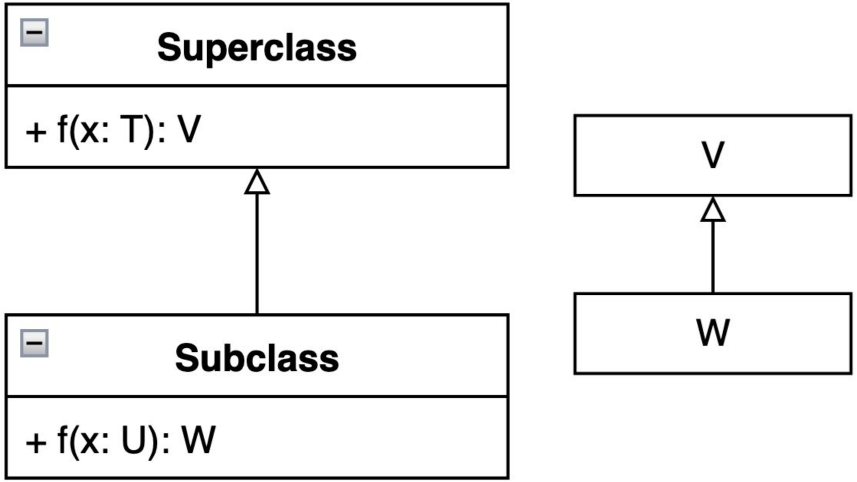
Check 2

In a **subclass**, the return type of a method should either **match** the return type in the corresponding method of the superclass or **be a subtype** of the return type in the superclass method.

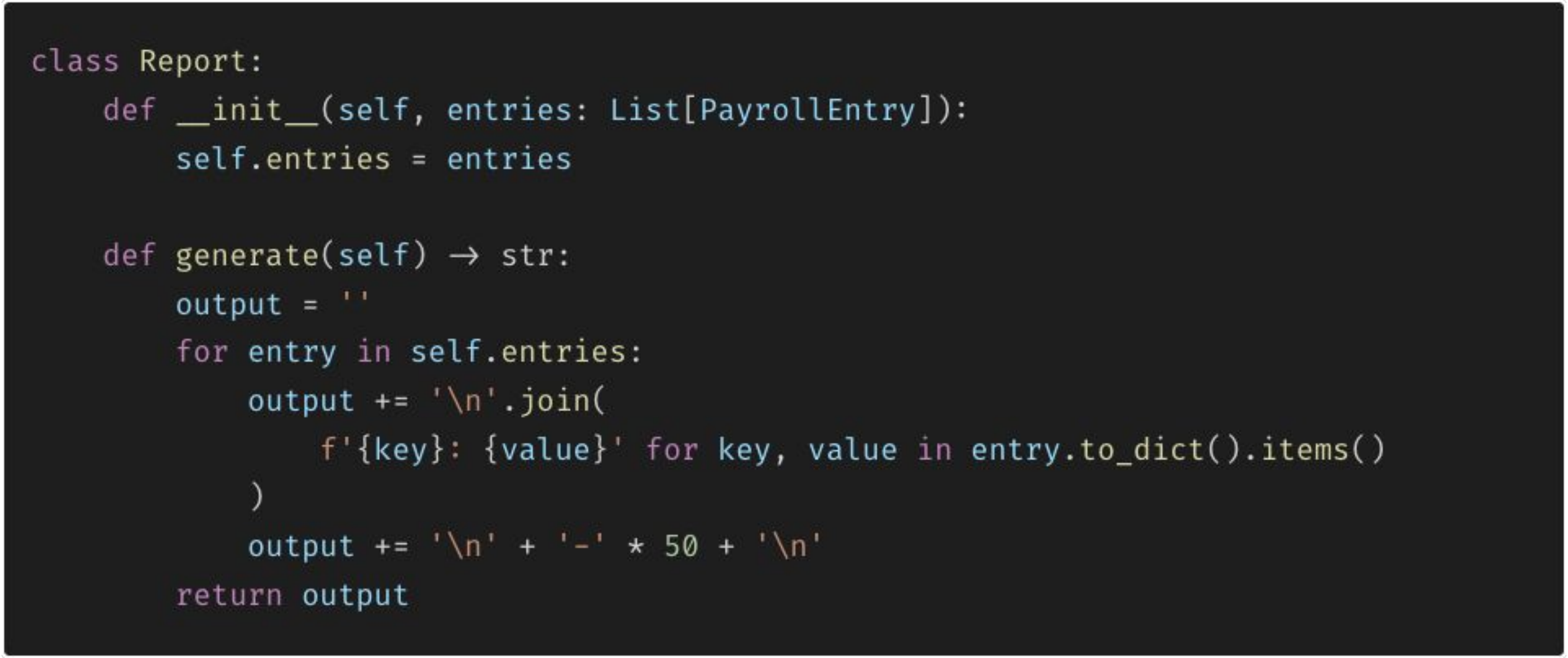


Check 2

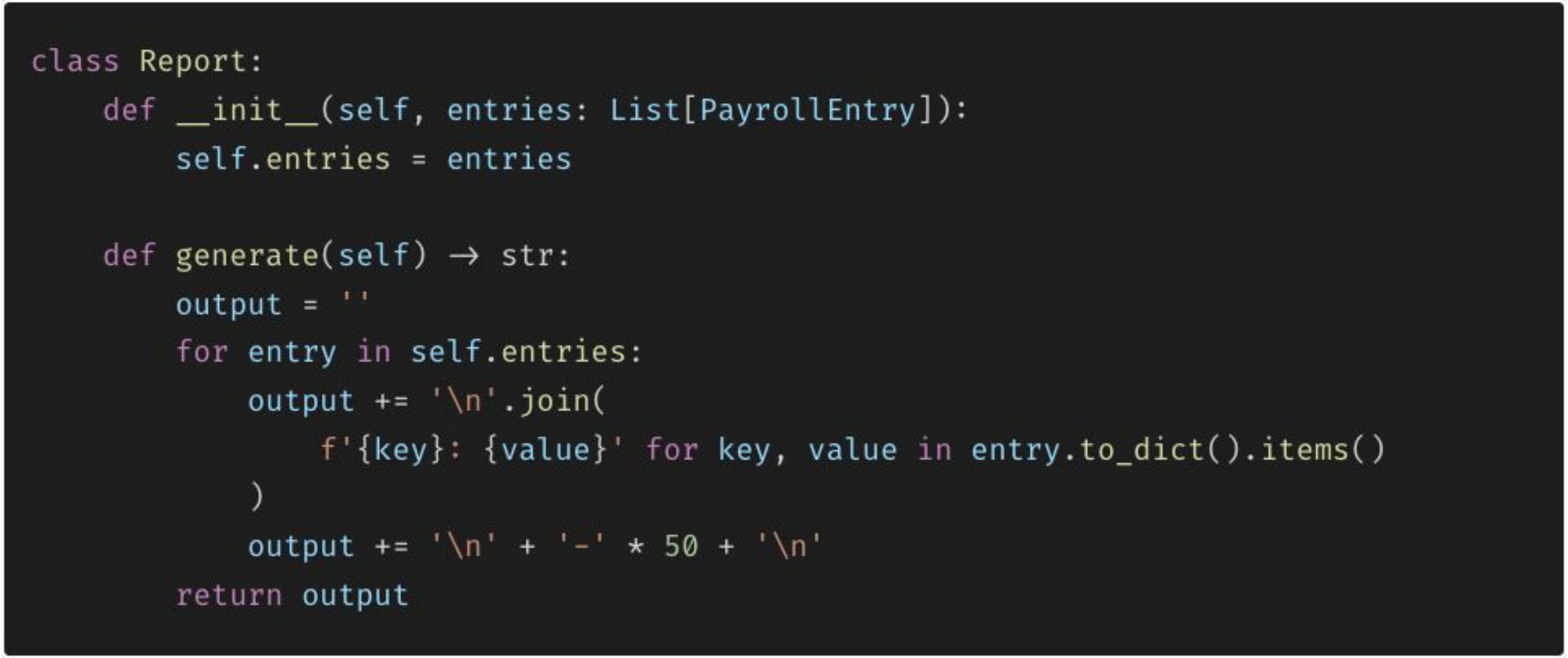
In a **subclass**, the return type of a method should either **match** the return type in the corresponding method of the superclass or **be a subtype** of the return type in the superclass method.



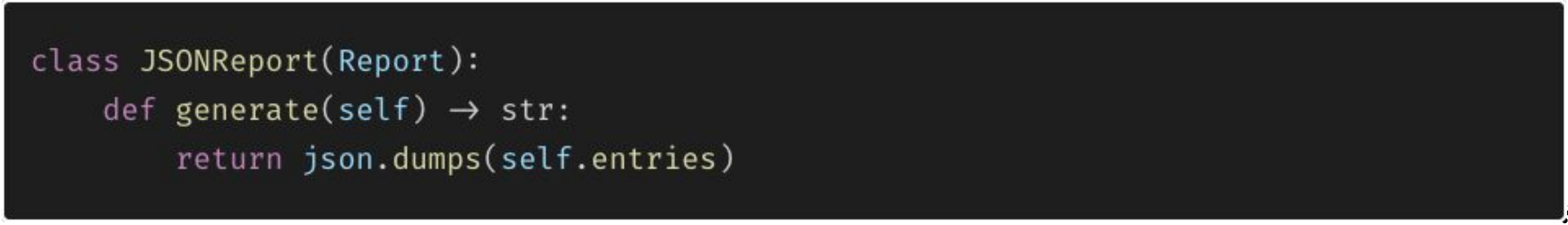
Covariance



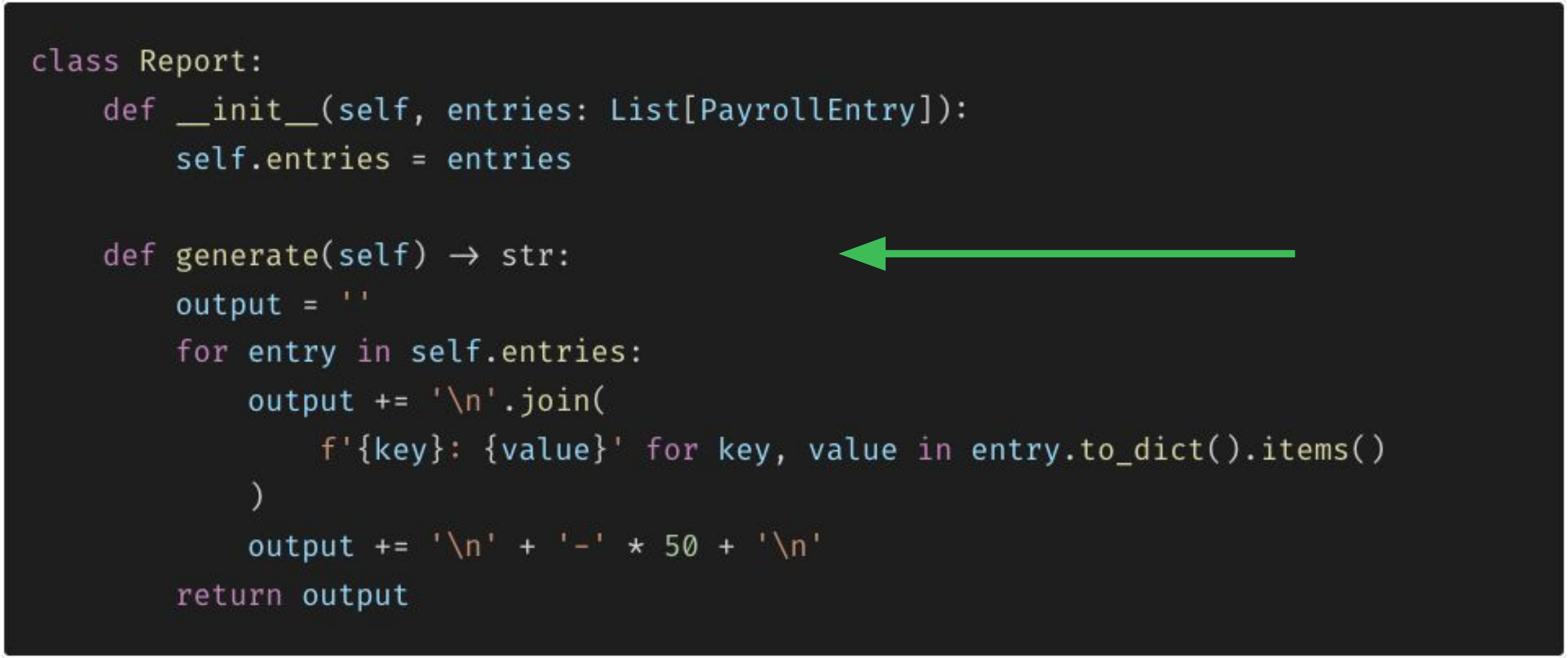
```python
class Report:
    def __init__(self, entries: List[PayrollEntry]):
        self.entries = entries

    def generate(self) -> str:
        output = ''
        for entry in self.entries:
            output += '\n'.join(
                f'{key}: {value}' for key, value in entry.to_dict().items()
            )
            output += '\n' + '-' * 50 + '\n'
        return output
```



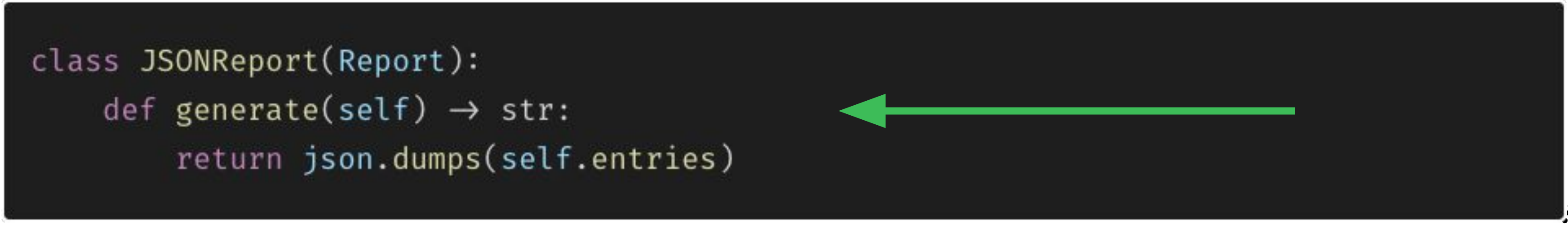
```python
class Report:
    def __init__(self, entries: List[PayrollEntry]):
        self.entries = entries

    def generate(self) -> str:
        output = ''
        for entry in self.entries:
            output += '\n'.join(
                f'{key}: {value}' for key, value in entry.to_dict().items()
            )
            output += '\n' + '-' * 50 + '\n'
        return output
```



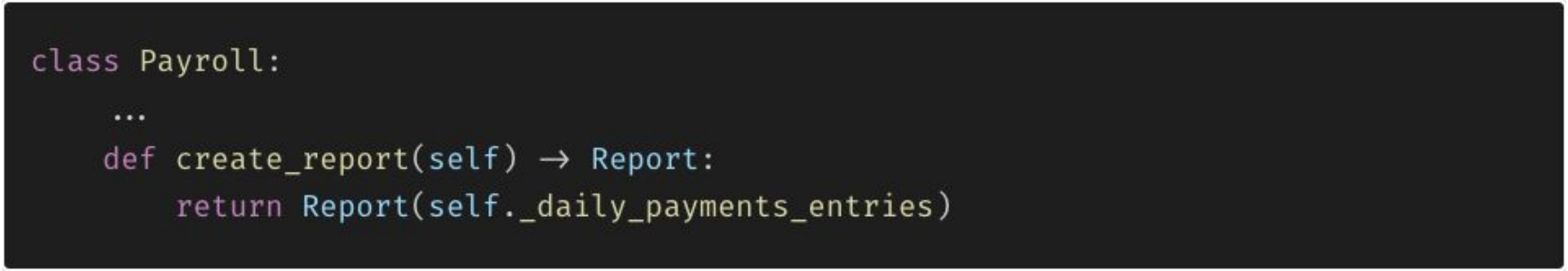
```python
class JSONReport(Report):
    def generate(self) -> str:
        return json.dumps(self.entries)
```



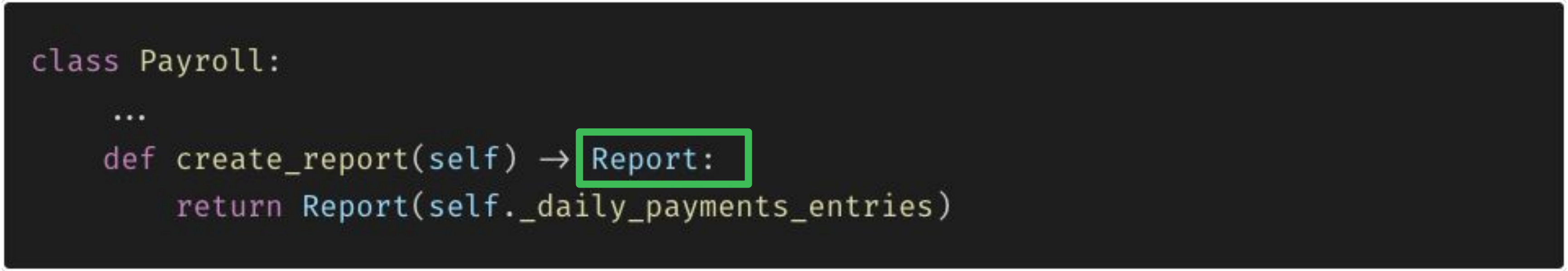
```python
class Report:
    def __init__(self, entries: List[PayrollEntry]):
        self.entries = entries

    def generate(self) -> str:
        output = ''
        for entry in self.entries:
            output += '\n'.join(
                f'{key}: {value}' for key, value in entry.to_dict().items()
            )
            output += '\n' + '-' * 50 + '\n'
        return output
```



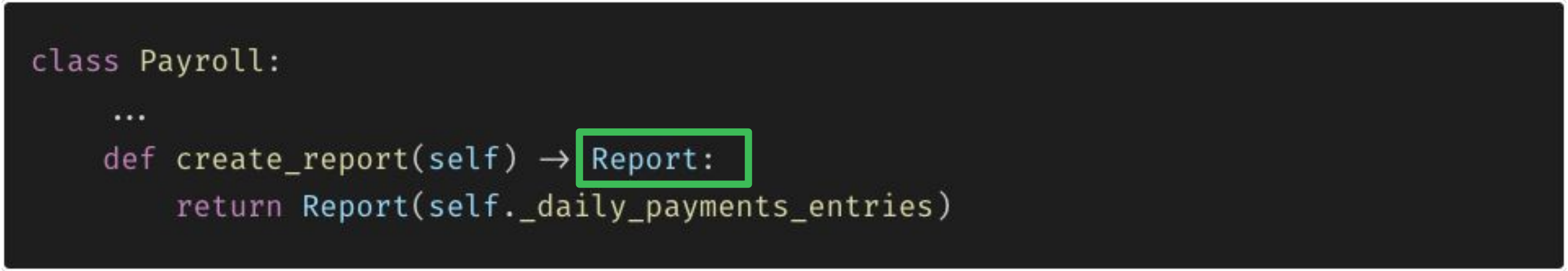
```python
class JSONReport(Report):
    def generate(self) -> str:
        return json.dumps(self.entries)
```



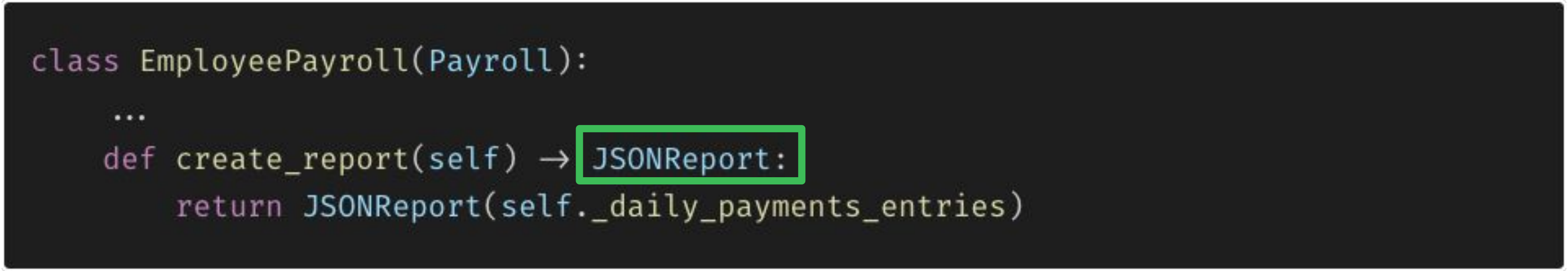
```python
class Payroll:
    ...
    def create_report(self) -> Report:
        return Report(self._daily_payments_entries)
```



```python
class Payroll:
    ...
    def create_report(self) -> Report:
        return Report(self._daily_payments_entries)
```



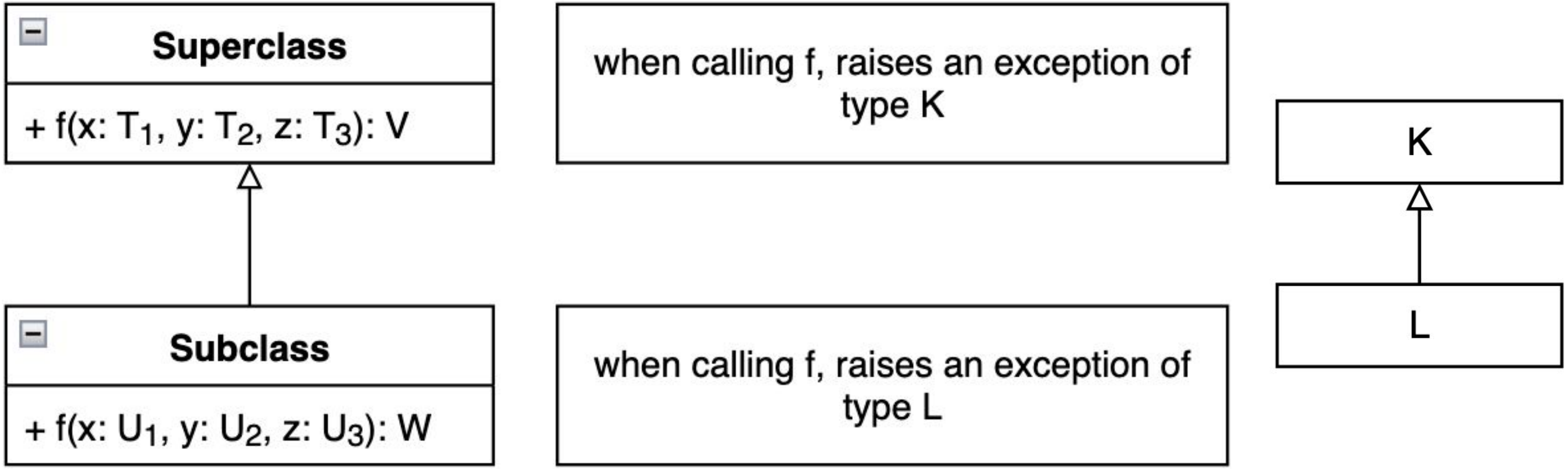
```python
class Payroll:
    ...
    def create_report(self) -> Report:
        return Report(self._daily_payments_entries)
```

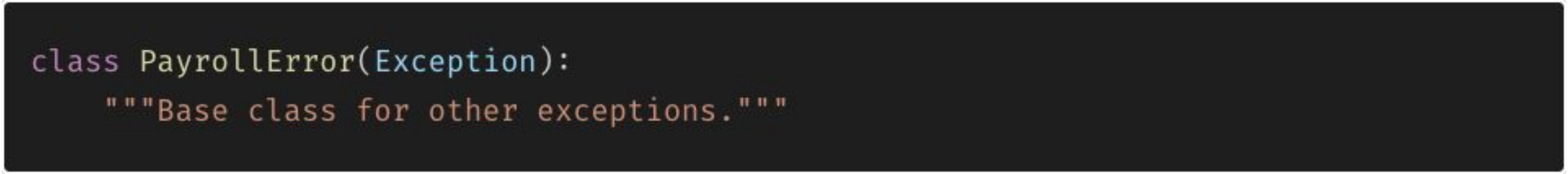


```python
class EmployeePayroll(Payroll):
    ...
    def create_report(self) -> JSONReport:
        return JSONReport(self._daily_payments_entries)
```

Check 3

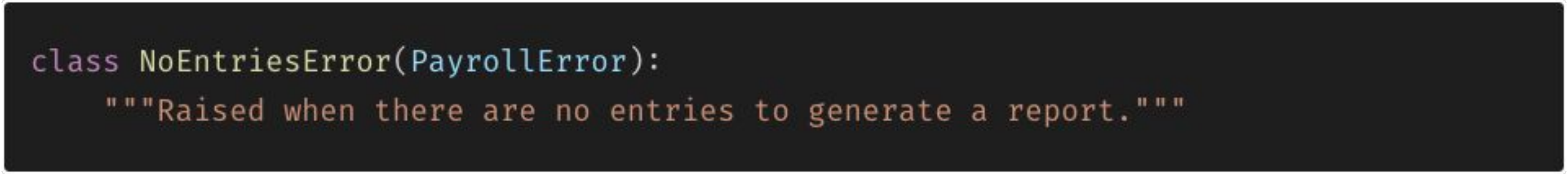
In a **subclass**, the types of exceptions thrown by a method should either **match** the types of exceptions that the corresponding base method in the **superclass** is already able to throw, or **be subtypes** of those exceptions.


Superclass
+ f(x: T_1, y: T_2, z: T_3): V
when calling f, raises an exception of type K
K
Subclass
+ f(x: U_1, y: U_2, z: U_3): W
when calling f, raises an exception of type L
L

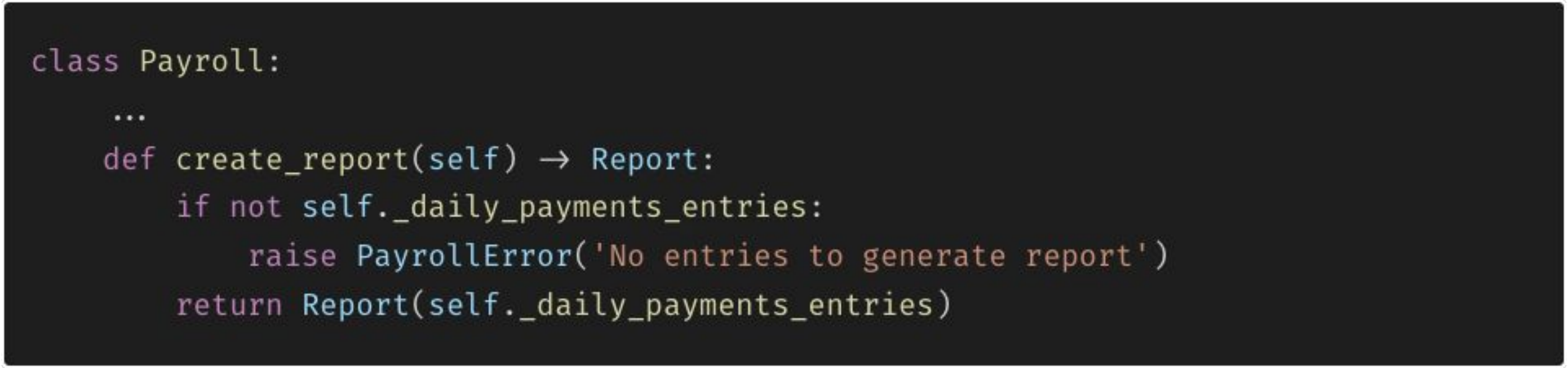


```python
class PayrollError(Exception):
    """Base class for other exceptions."""
```

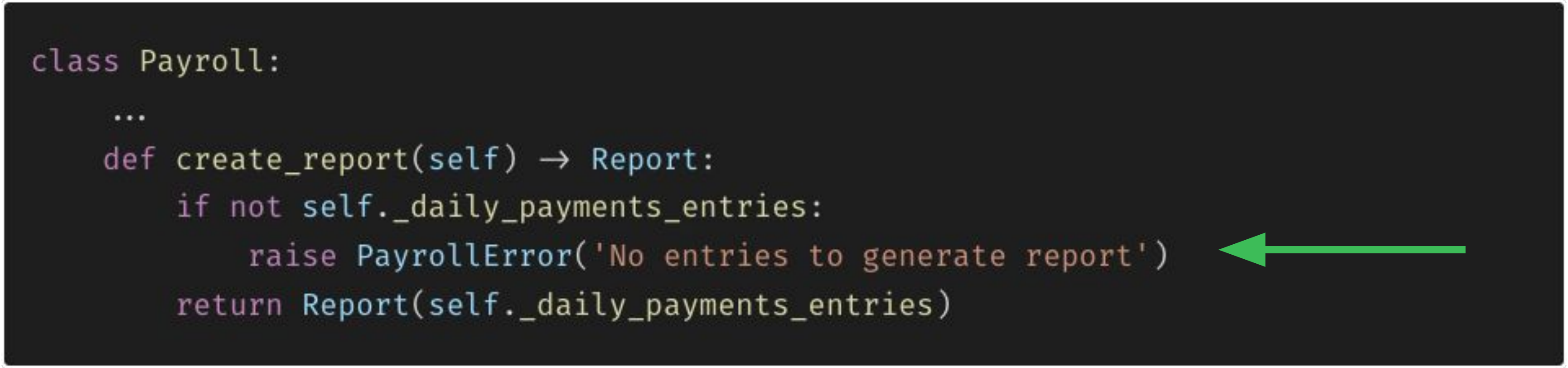
```python
class PayrollError(Exception):
    """Base class for other exceptions."""
```



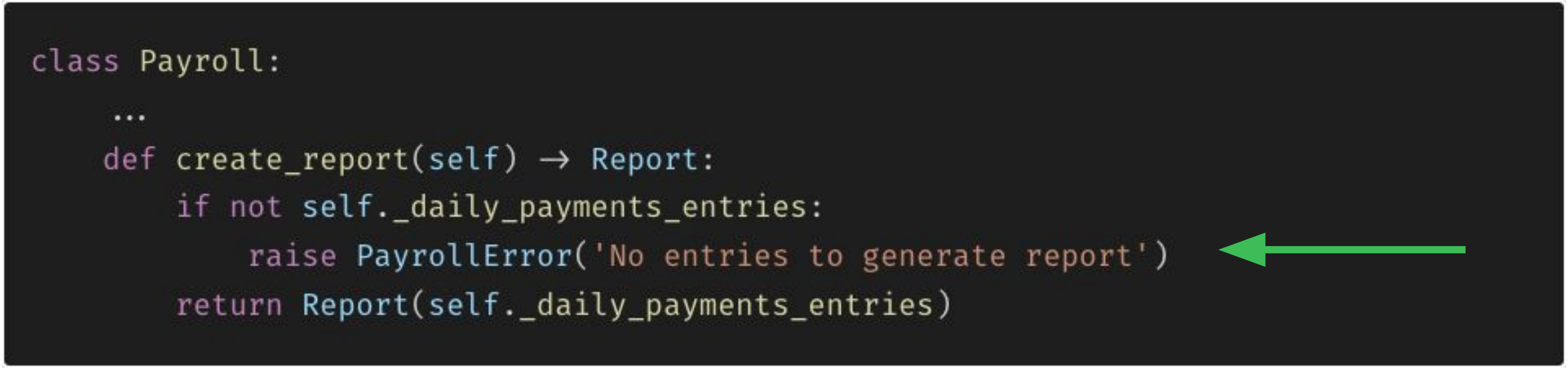
```python
class NoEntriesError(PayrollError):
    """Raised when there are no entries to generate a report."""
```



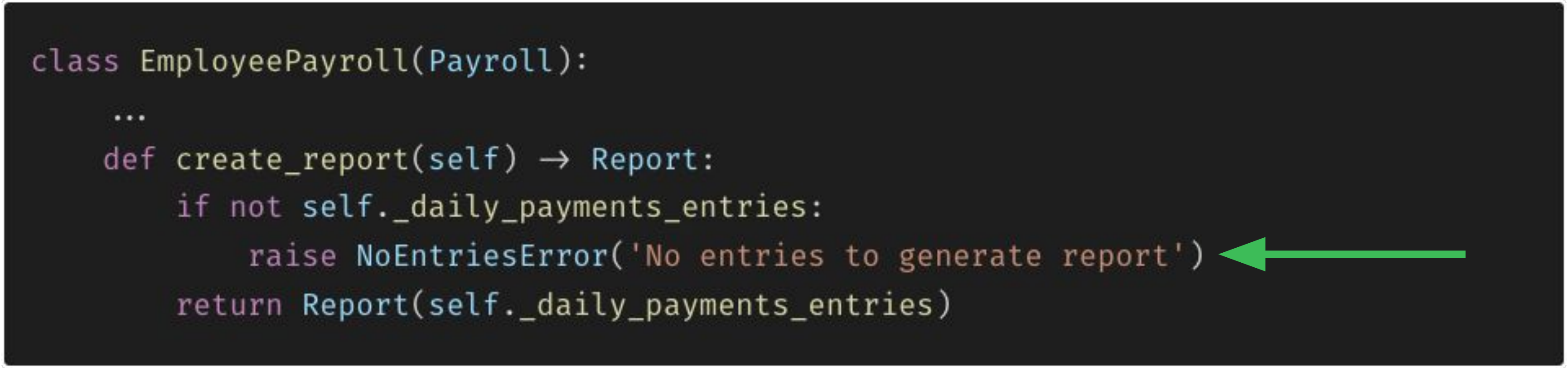
```python
class Payroll:
    ...
    def create_report(self) -> Report:
        if not self._daily_payments_entries:
            raise PayrollError('No entries to generate report')
        return Report(self._daily_payments_entries)
```



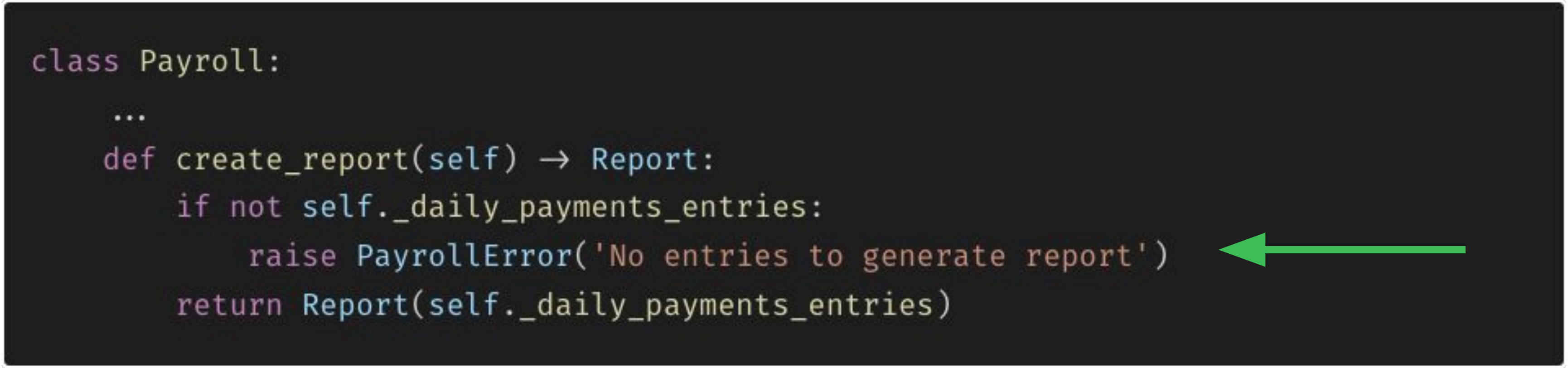
```python
class Payroll:
    ...
    def create_report(self) -> Report:
        if not self._daily_payments_entries:
            raise PayrollError('No entries to generate report')
        return Report(self._daily_payments_entries)
```



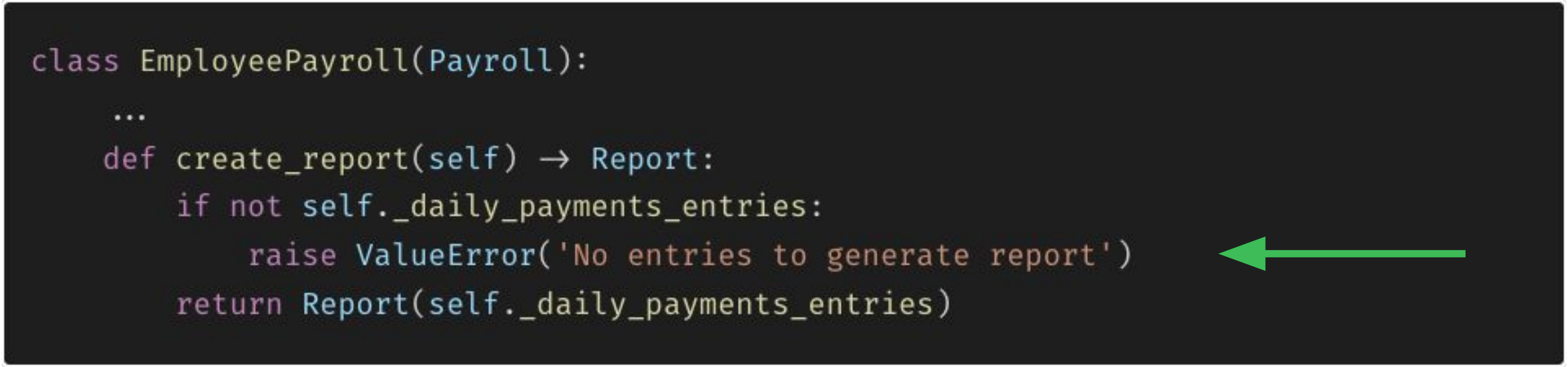
```python
class Payroll:
    ...
    def create_report(self) -> Report:
        if not self._daily_payments_entries:
            raise PayrollError('No entries to generate report')
        return Report(self._daily_payments_entries)
```



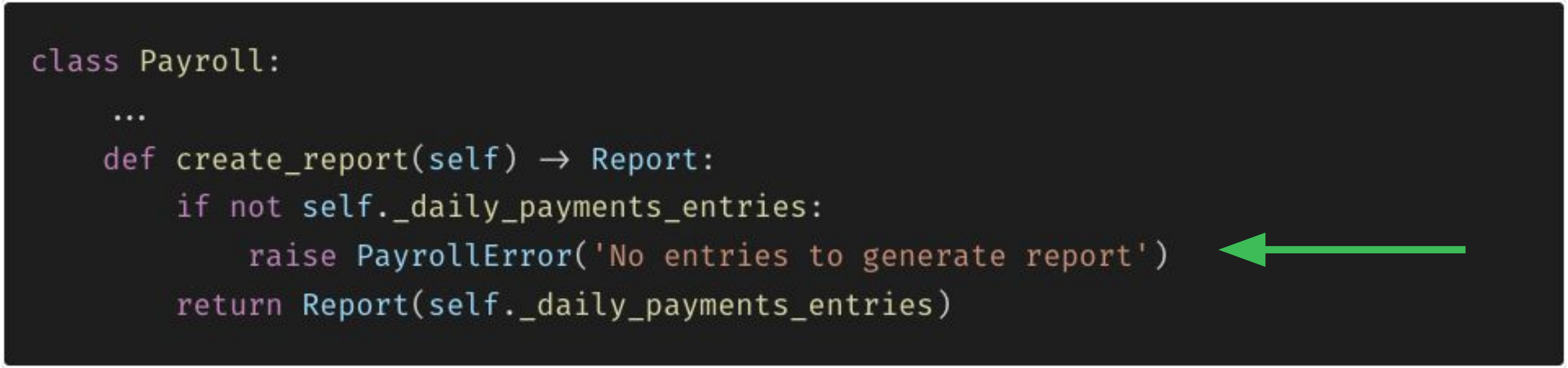
```python
class EmployeePayroll(Payroll):
    ...
    def create_report(self) -> Report:
        if not self._daily_payments_entries:
            raise NoEntriesError('No entries to generate report')
        return Report(self._daily_payments_entries)
```



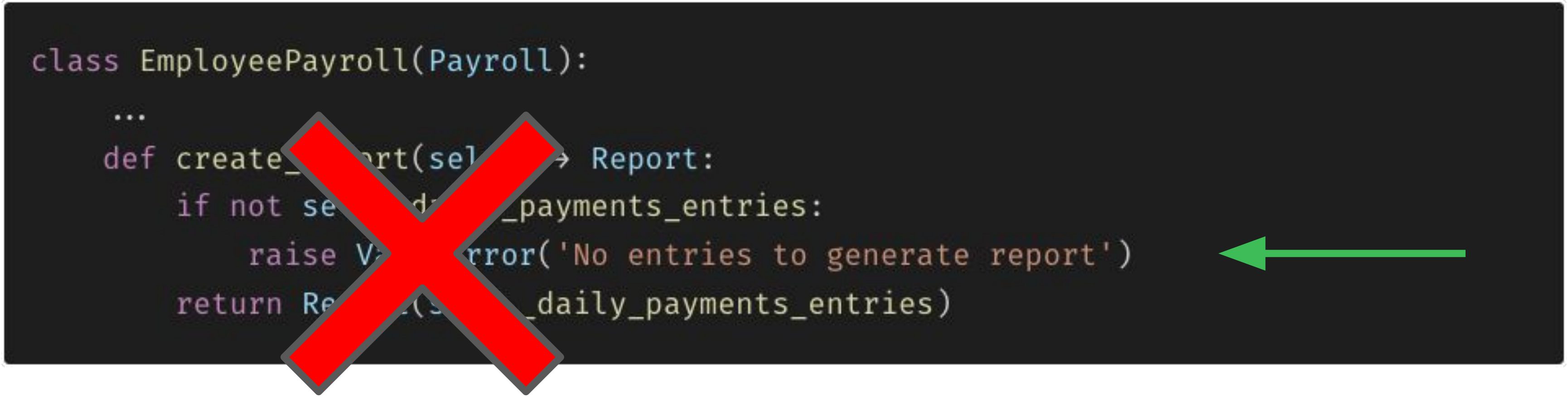
```python
class Payroll:
    ...
    def create_report(self) -> Report:
        if not self._daily_payments_entries:
            raise PayrollError('No entries to generate report')
        return Report(self._daily_payments_entries)
```



```python
class EmployeePayroll(Payroll):
    ...
    def create_report(self) -> Report:
        if not self._daily_payments_entries:
            raise ValueError('No entries to generate report')
        return Report(self._daily_payments_entries)
```



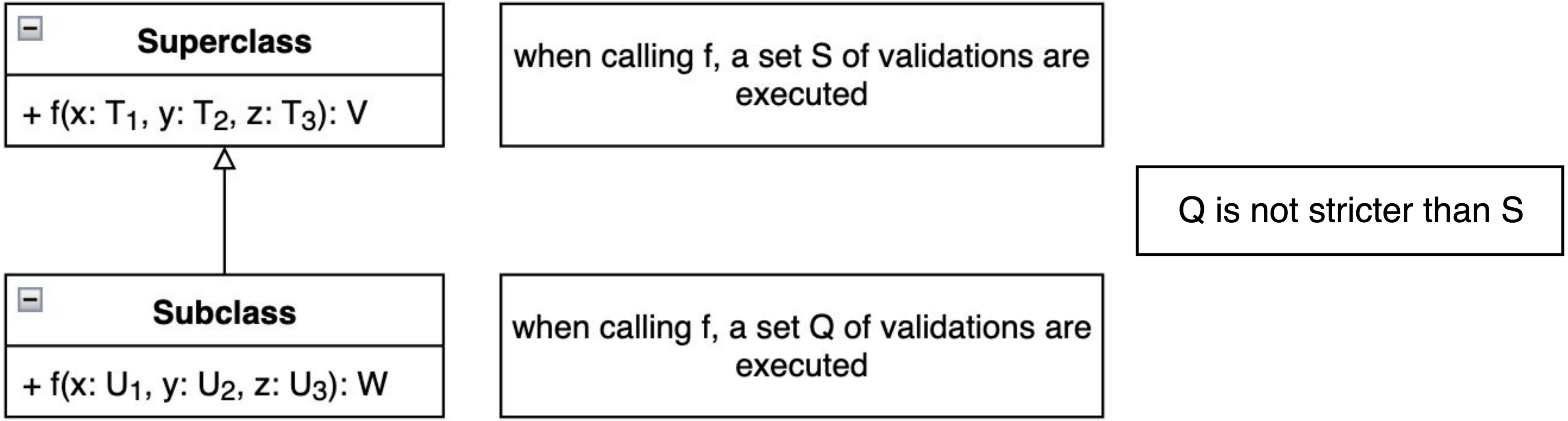
```
class Payroll:
    ...
    def create_report(self) -> Report:
        if not self._daily_payments_entries:
            raise PayrollError('No entries to generate report')
        return Report(self._daily_payments_entries)
```



```
class EmployeePayroll(Payroll):
    ...
    def create_report(self) -> Report:
        if not self._daily_payments_entries:
            raise ValueError('No entries to generate report')
        return Report(self._daily_payments_entries)
```

Check 4

A **subclass should not impose stricter preconditions** than those defined by its **superclass**.



Superclass

+ $f(x: T_1, y: T_2, z: T_3): V$

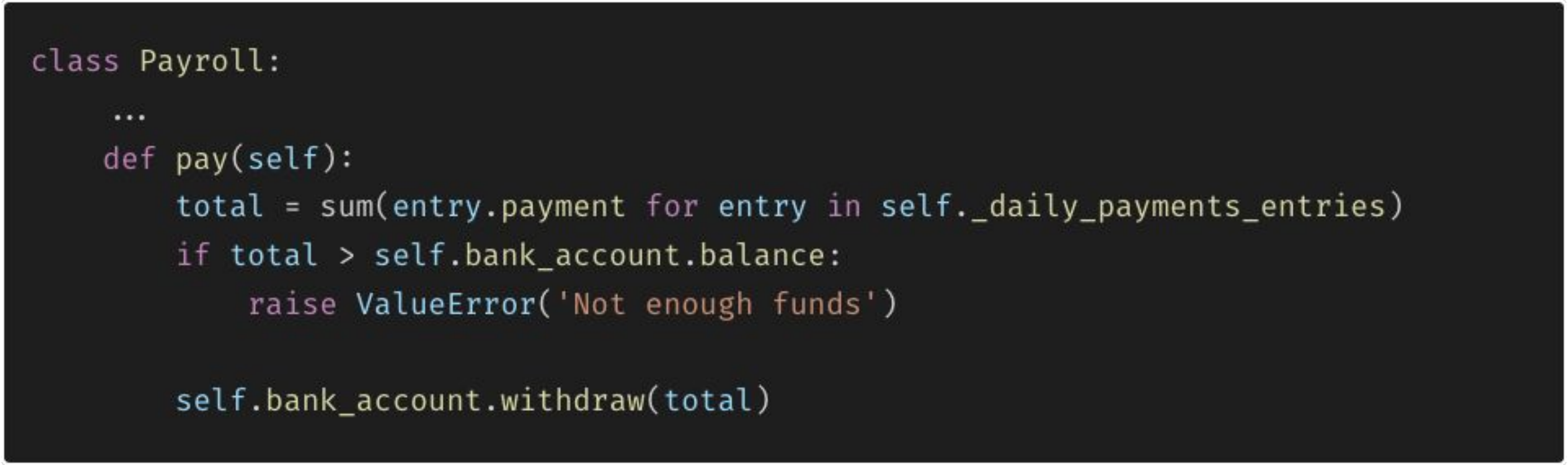
when calling f, a set S of validations are executed

Subclass

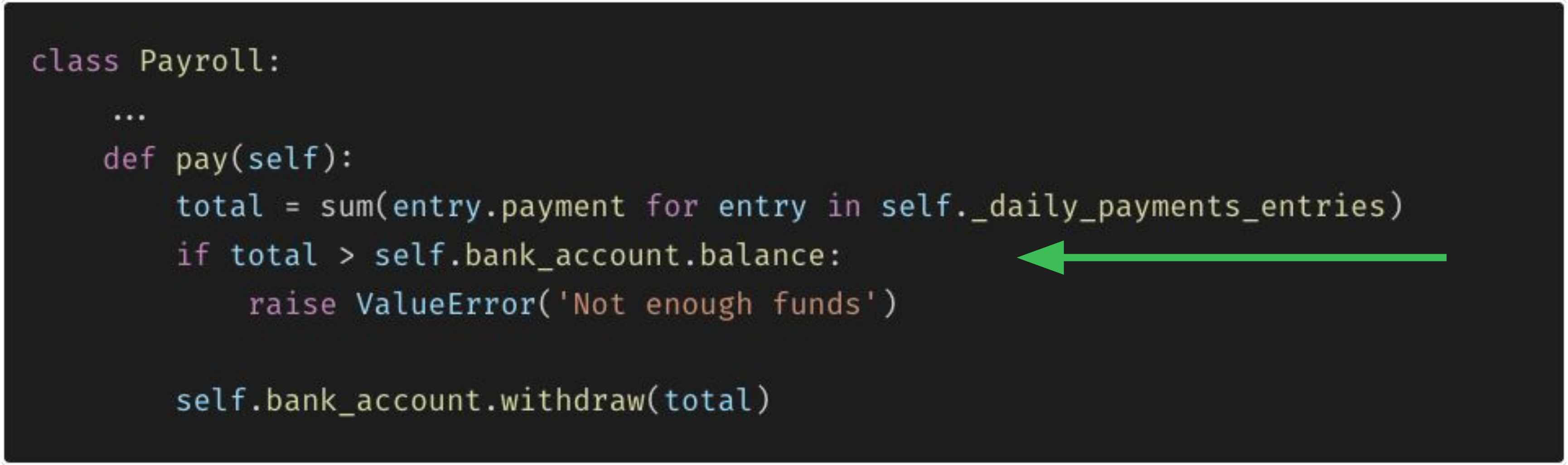
+ $f(x: U_1, y: U_2, z: U_3): W$

when calling f, a set Q of validations are executed

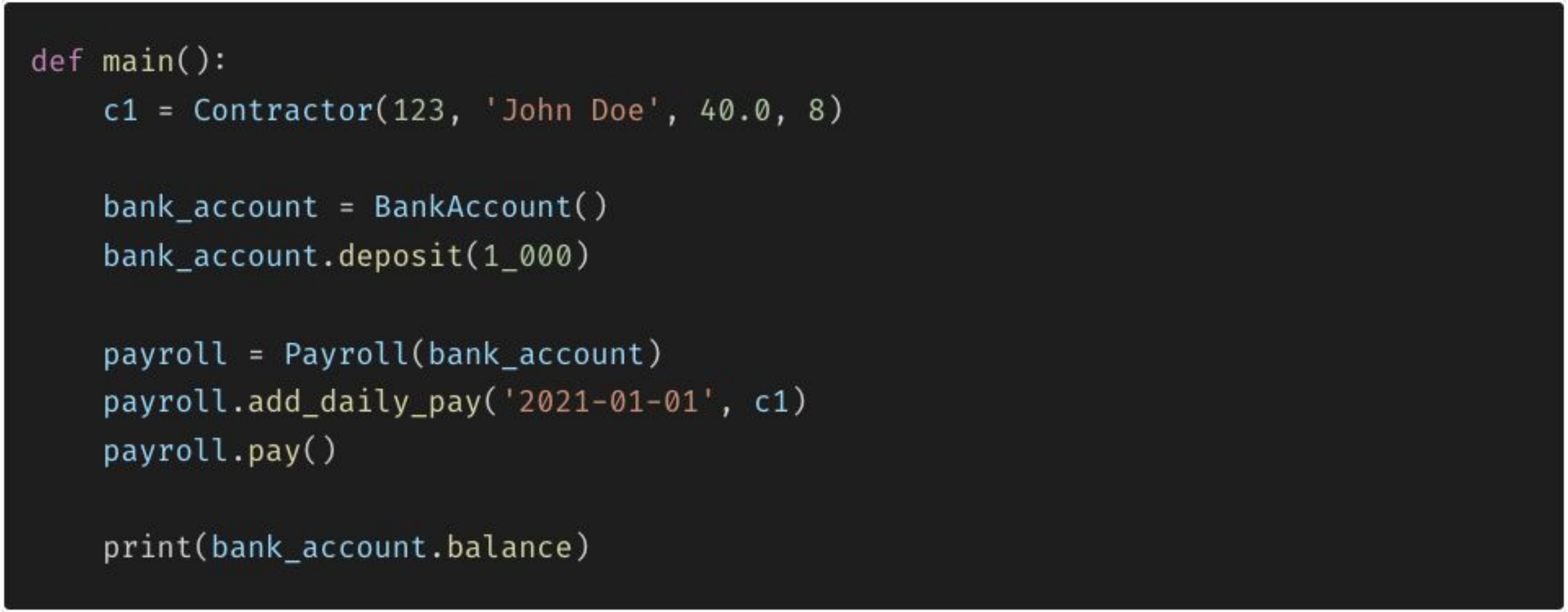
Q is not stricter than S



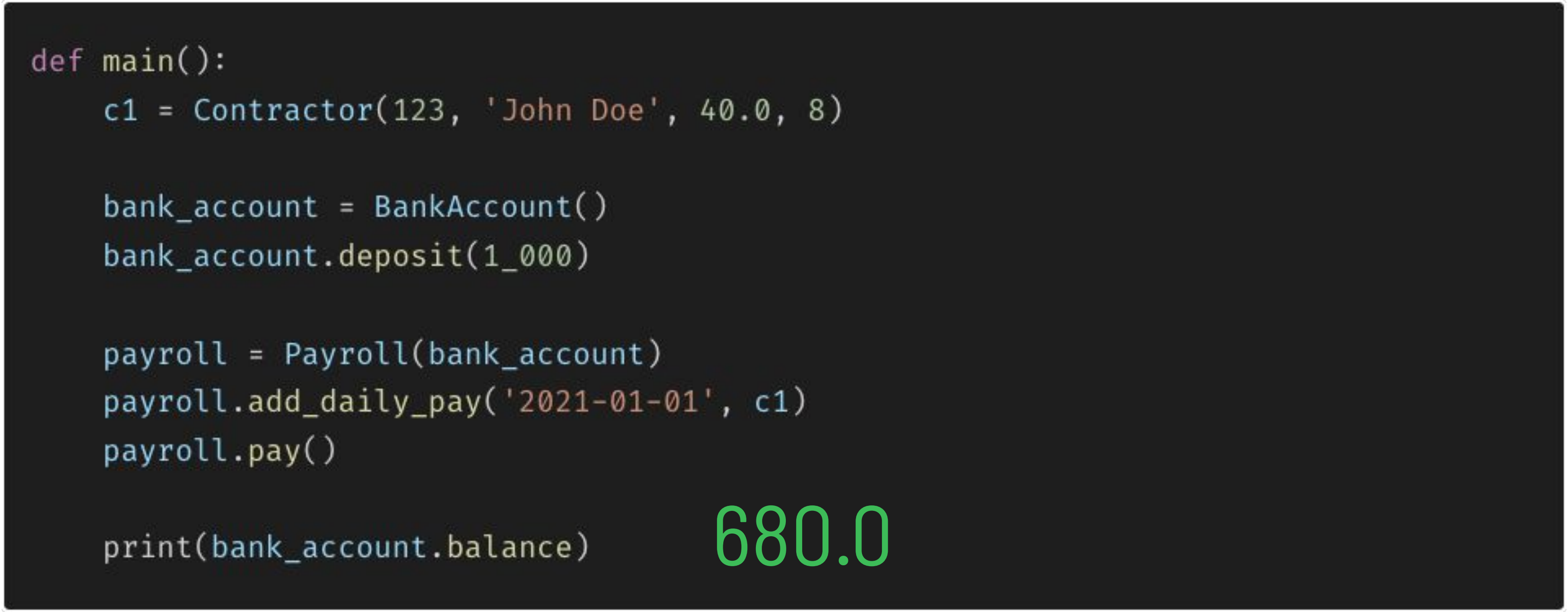
```python
class Payroll:
    ...
    def pay(self):
        total = sum(entry.payment for entry in self._daily_payments_entries)
        if total > self.bank_account.balance:
            raise ValueError('Not enough funds')

        self.bank_account.withdraw(total)
```



```python
class Payroll:
    ...
    def pay(self):
        total = sum(entry.payment for entry in self._daily_payments_entries)
        if total > self.bank_account.balance:
            raise ValueError('Not enough funds')

        self.bank_account.withdraw(total)
```

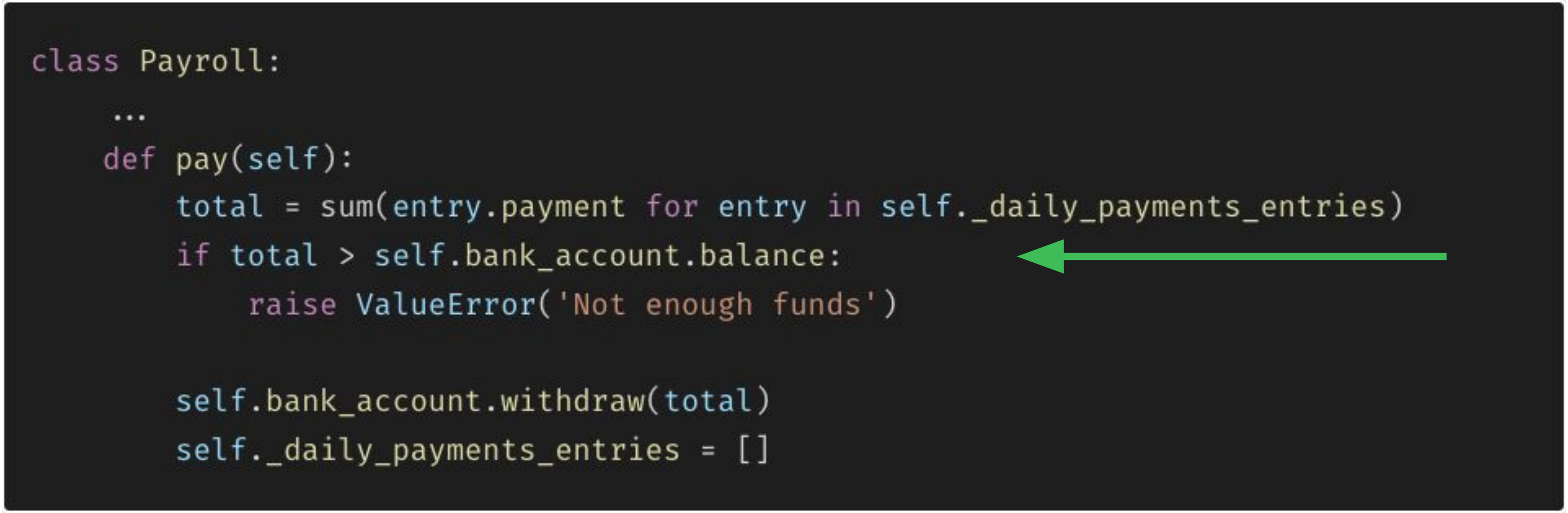


```python
def main():
    c1 = Contractor(123, 'John Doe', 40.0, 8)

    bank_account = BankAccount()
    bank_account.deposit(1_000)

    payroll = Payroll(bank_account)
    payroll.add_daily_pay('2021-01-01', c1)
    payroll.pay()

    print(bank_account.balance)
```

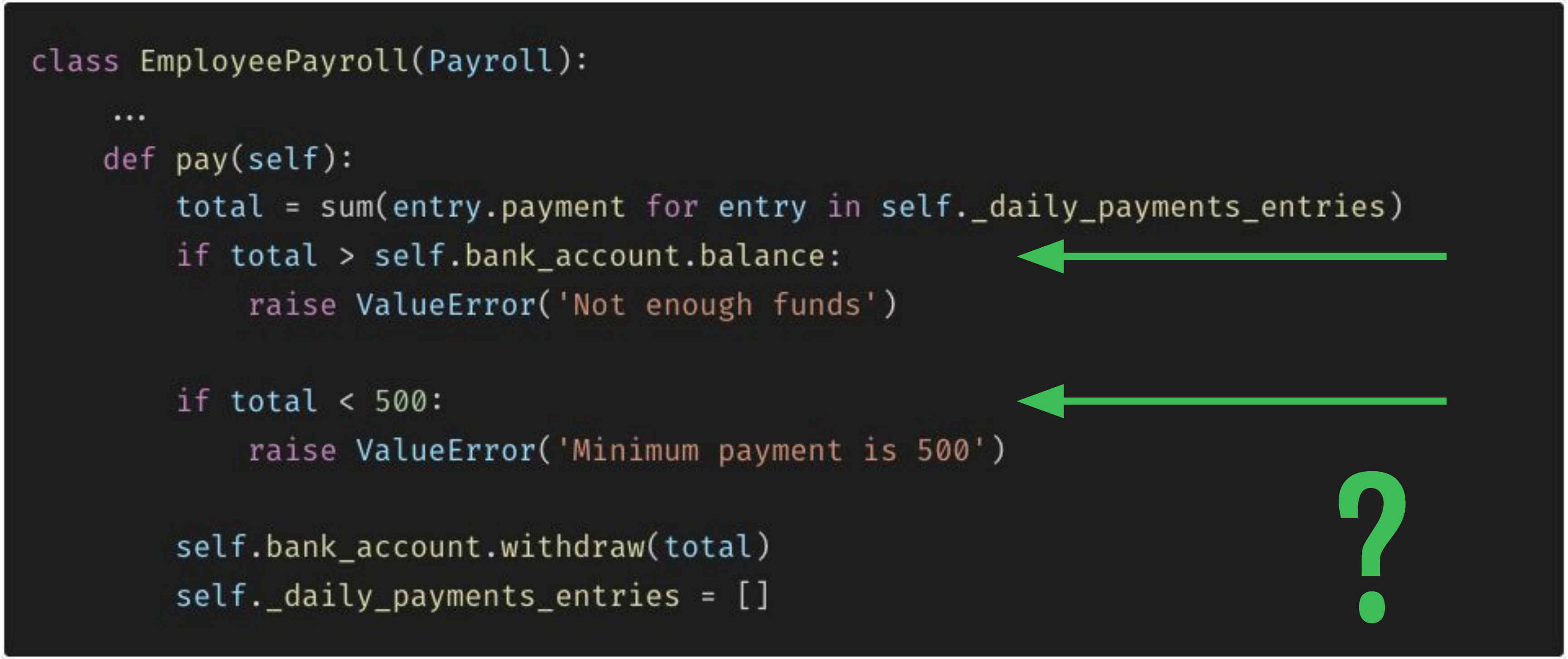


```python
def main():
    c1 = Contractor(123, 'John Doe', 40.0, 8)

    bank_account = BankAccount()
    bank_account.deposit(1_000)

    payroll = Payroll(bank_account)
    payroll.add_daily_pay('2021-01-01', c1)
    payroll.pay()

    print(bank_account.balance)
```

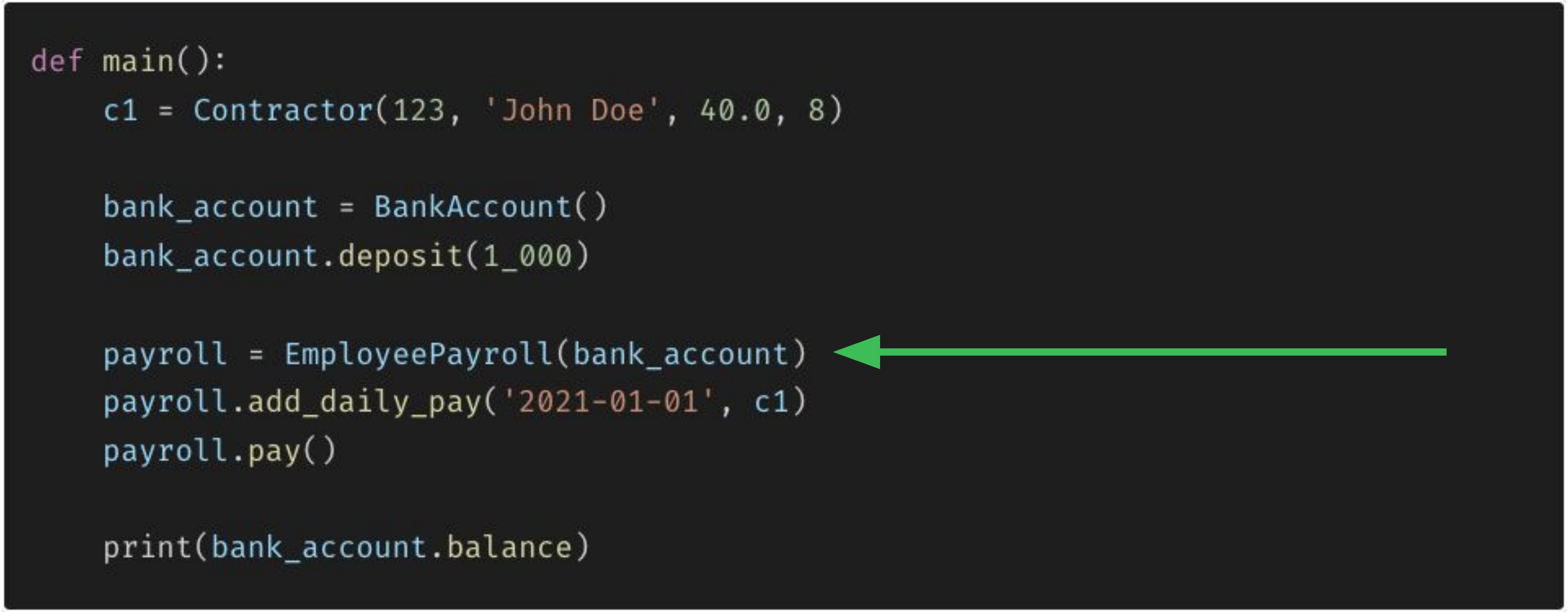
```
680.0
```



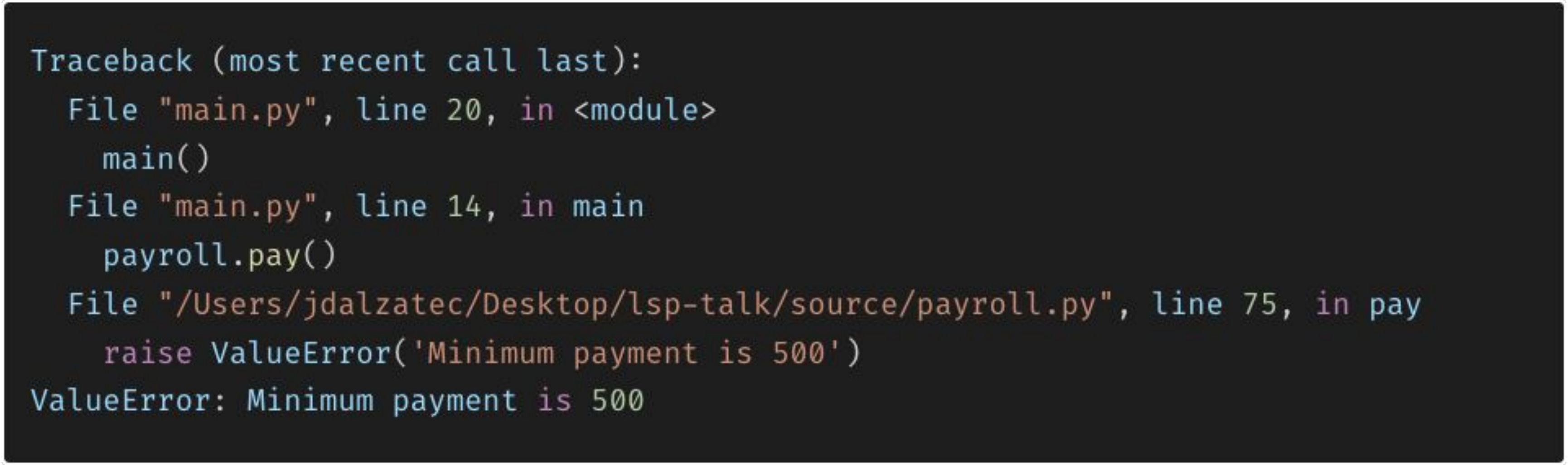
```python
class Payroll:
    ...
    def pay(self):
        total = sum(entry.payment for entry in self._daily_payments_entries)
        if total > self.bank_account.balance:
            raise ValueError('Not enough funds')

        self.bank_account.withdraw(total)
        self._daily_payments_entries = []
```



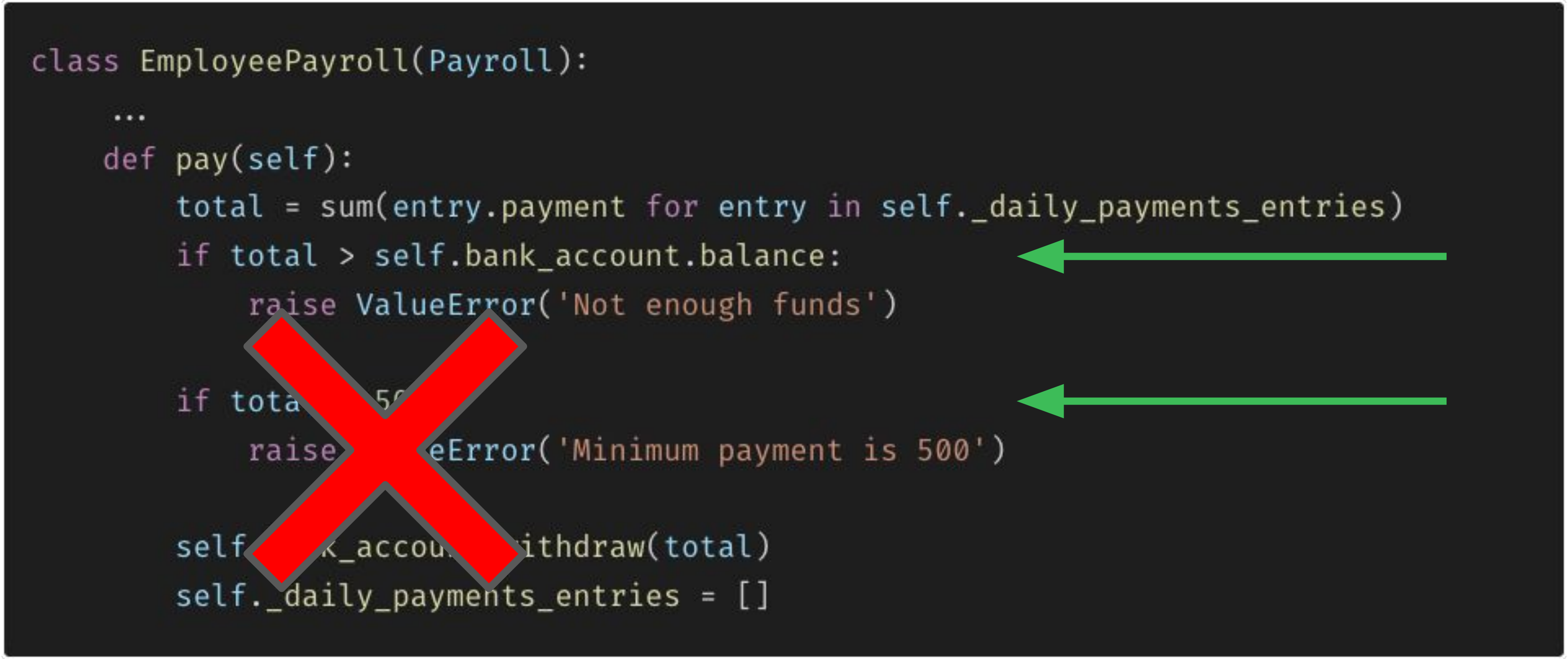
```python
class EmployeePayroll(Payroll):
    ...
    def pay(self):
        total = sum(entry.payment for entry in self._daily_payments_entries)
        if total > self.bank_account.balance:
            raise ValueError('Not enough funds')

        if total < 500:
            raise ValueError('Minimum payment is 500')

        self.bank_account.withdraw(total)
        self._daily_payments_entries = []
```



```python
def main():
    c1 = Contractor(123, 'John Doe', 40.0, 8)

    bank_account = BankAccount()
    bank_account.deposit(1_000)

    payroll = EmployeePayroll(bank_account)
    payroll.add_daily_pay('2021-01-01', c1)
    payroll.pay()

    print(bank_account.balance)
```



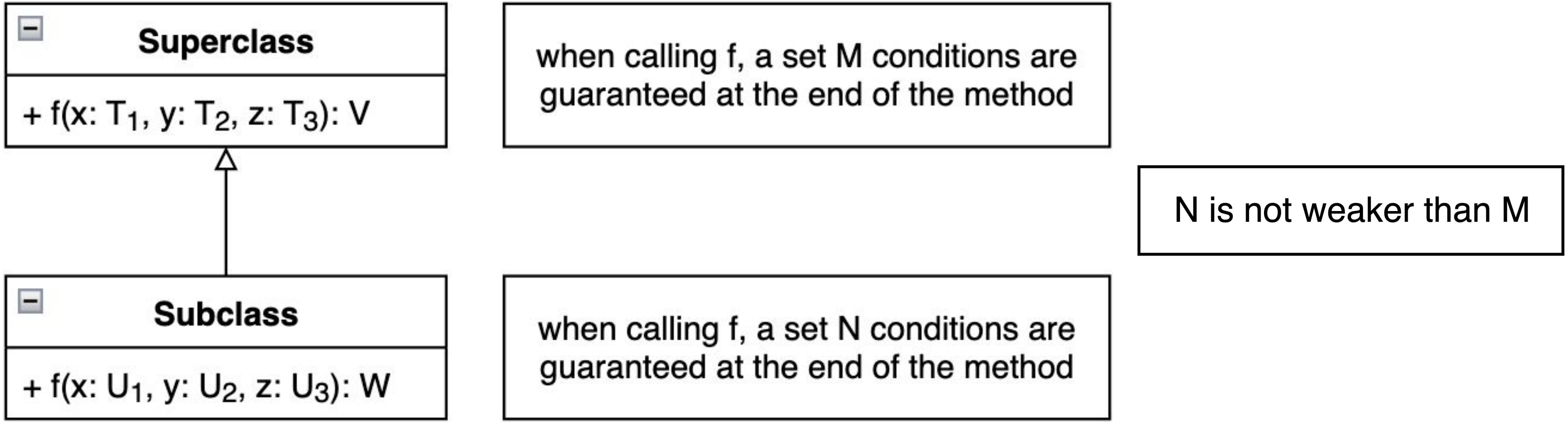
```
Traceback (most recent call last):
  File "main.py", line 20, in <module>
    main()
  File "main.py", line 14, in main
    payroll.pay()
  File "/Users/jdalzatec/Desktop/lsp-talk/source/payroll.py", line 75, in pay
    raise ValueError('Minimum payment is 500')
ValueError: Minimum payment is 500
```



```
class EmployeePayroll(Payroll):
    ...
    def pay(self):
        total = sum(entry.payment for entry in self._daily_payments_entries)
        if total > self.bank_account.balance:
            raise ValueError('Not enough funds')

        if tota[illegible] 5[illegible]
            raise [illegible]eError('Minimum payment is 500')

        self[illegible]k_accou[illegible]ithdraw(total)
        self._daily_payments_entries = []
```

Check 5

A **subclass should not relax or weaken the post-conditions** defined by its **superclass**.



Superclass

+ f(x: T_1, y: T_2, z: T_3): V

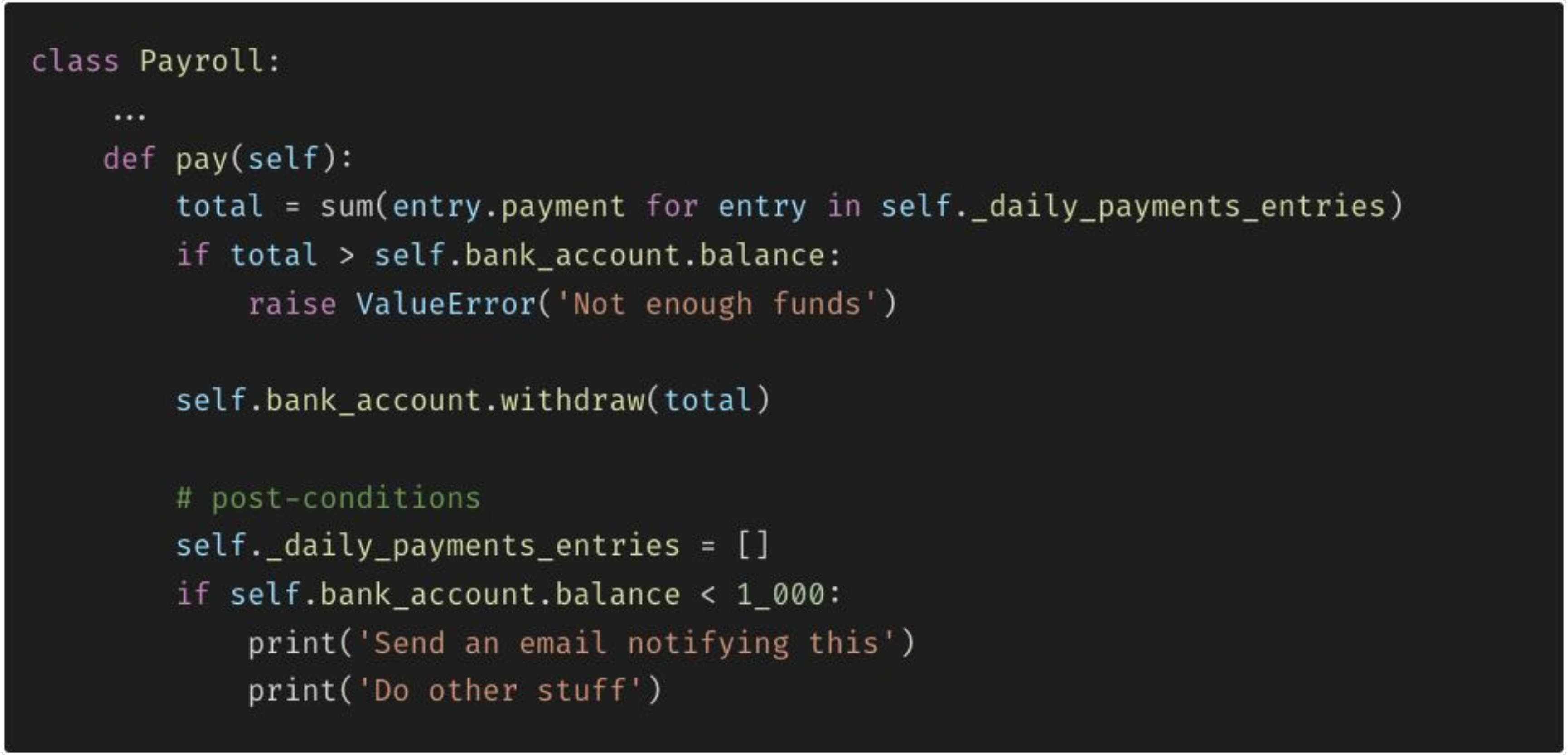
when calling f, a set M conditions are guaranteed at the end of the method

N is not weaker than M

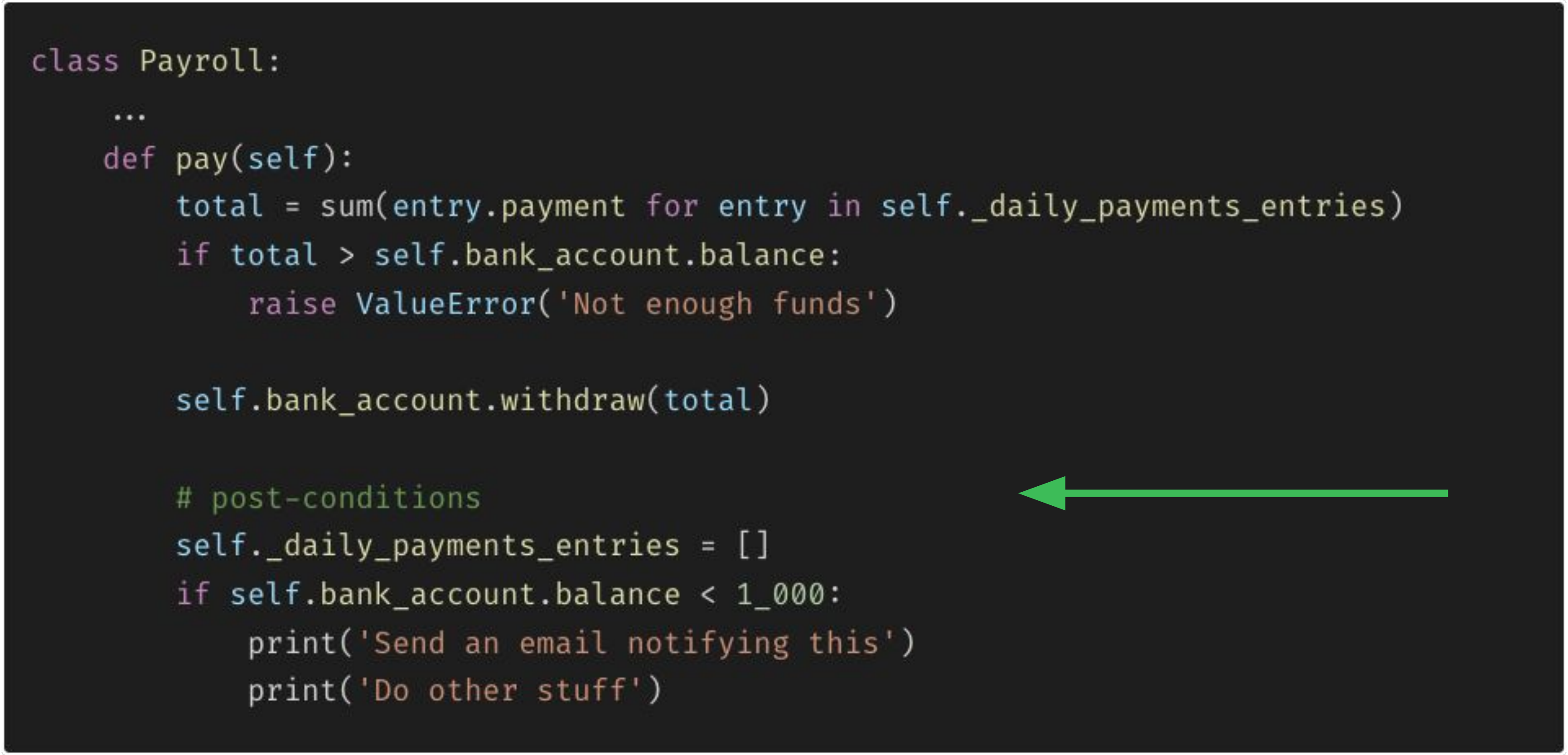
Subclass

+ f(x: U_1, y: U_2, z: U_3): W

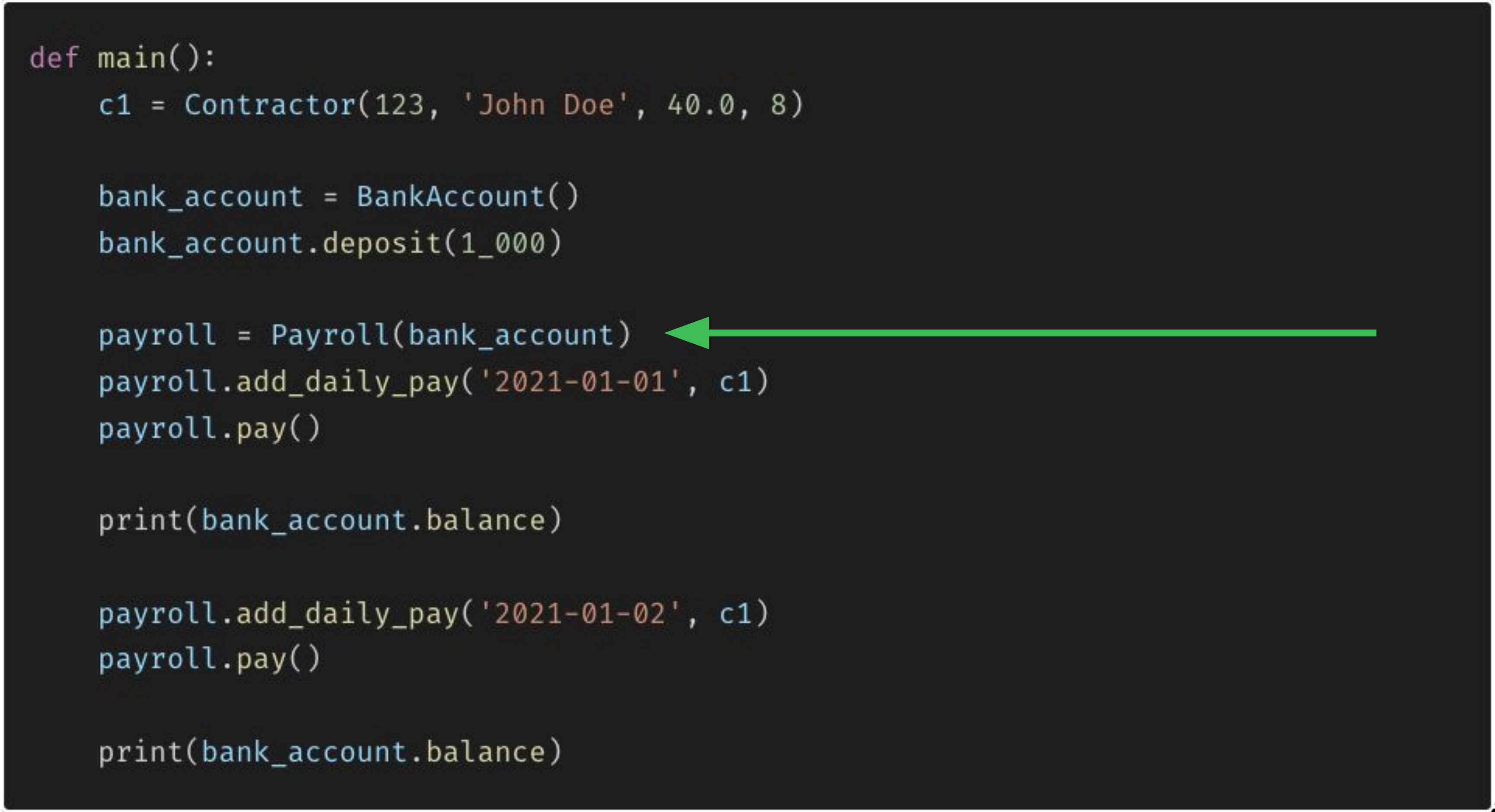
when calling f, a set N conditions are guaranteed at the end of the method



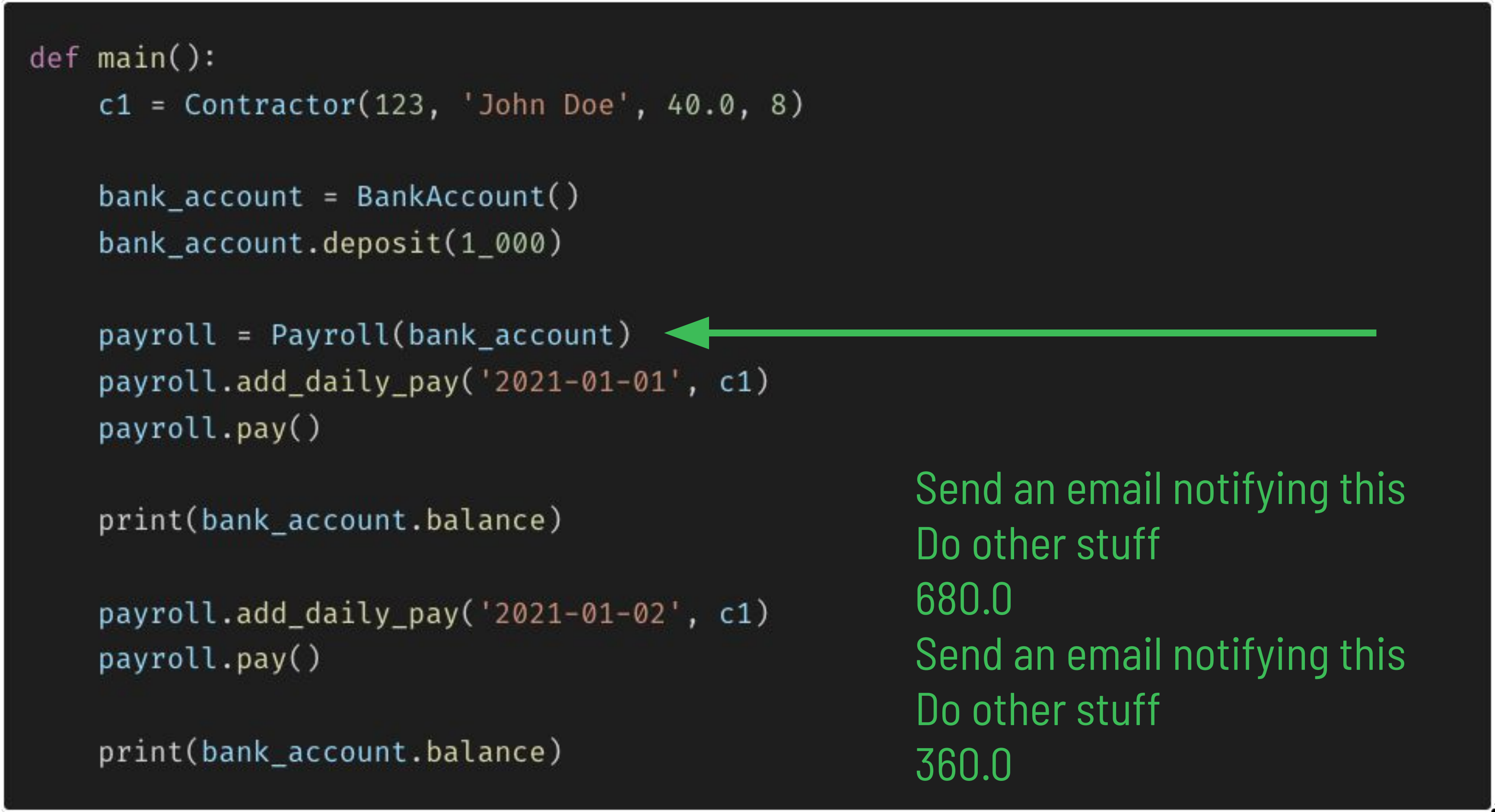
```python
class Payroll:
    ...
    def pay(self):
        total = sum(entry.payment for entry in self._daily_payments_entries)
        if total > self.bank_account.balance:
            raise ValueError('Not enough funds')

        self.bank_account.withdraw(total)

        # post-conditions
        self._daily_payments_entries = []
        if self.bank_account.balance < 1_000:
            print('Send an email notifying this')
            print('Do other stuff')
```



```python
class Payroll:
    ...
    def pay(self):
        total = sum(entry.payment for entry in self._daily_payments_entries)
        if total > self.bank_account.balance:
            raise ValueError('Not enough funds')

        self.bank_account.withdraw(total)

        # post-conditions
        self._daily_payments_entries = []
        if self.bank_account.balance < 1_000:
            print('Send an email notifying this')
            print('Do other stuff')
```

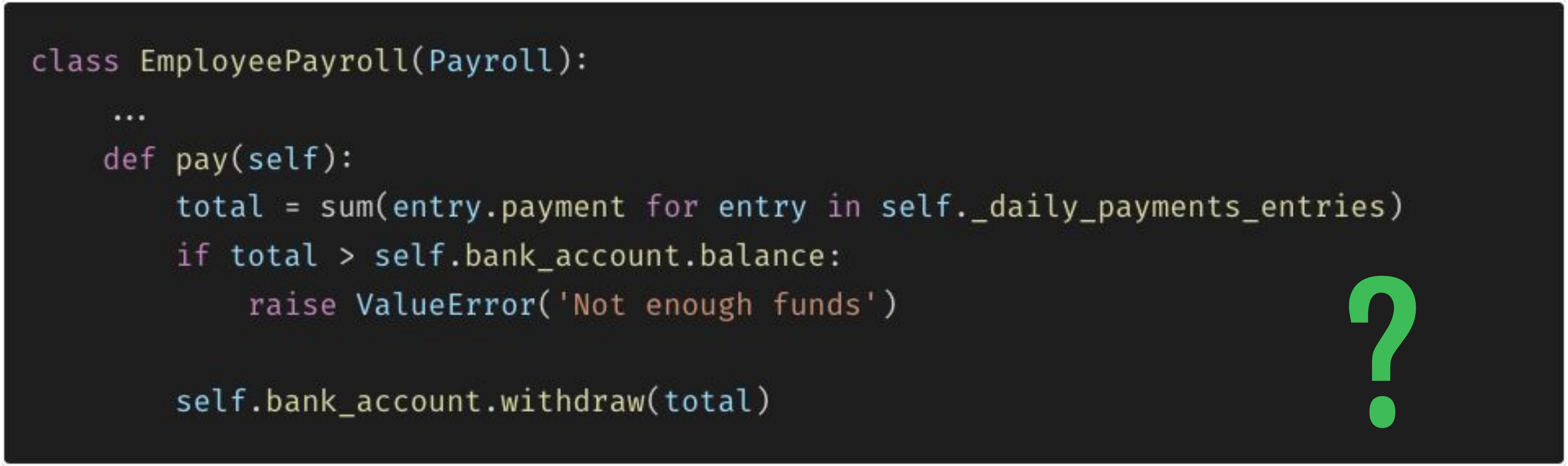


```python
def main():
    c1 = Contractor(123, 'John Doe', 40.0, 8)

    bank_account = BankAccount()
    bank_account.deposit(1_000)

    payroll = Payroll(bank_account)
    payroll.add_daily_pay('2021-01-01', c1)
    payroll.pay()

    print(bank_account.balance)

    payroll.add_daily_pay('2021-01-02', c1)
    payroll.pay()

    print(bank_account.balance)
```

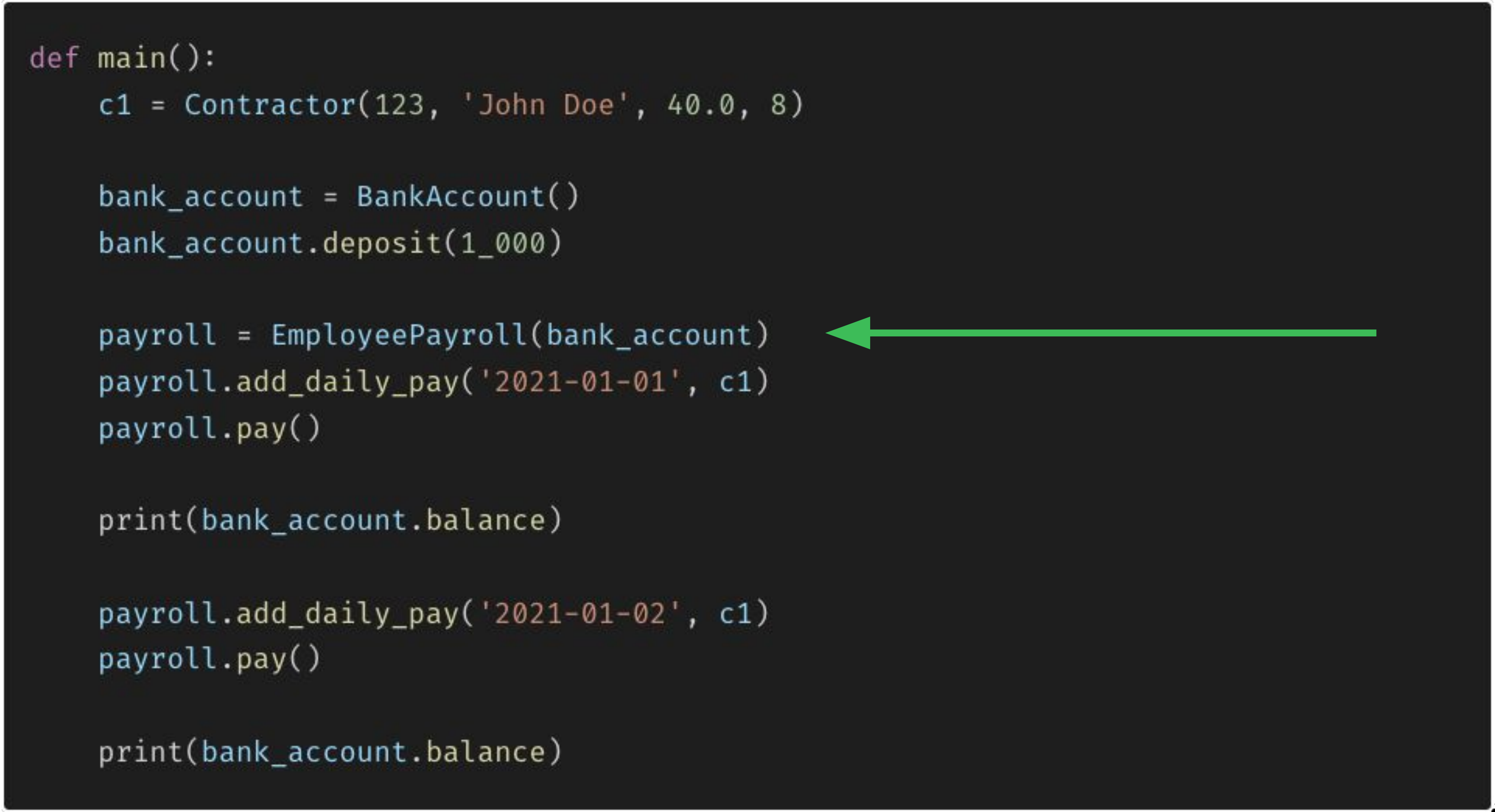


```python
def main():
    c1 = Contractor(123, 'John Doe', 40.0, 8)

    bank_account = BankAccount()
    bank_account.deposit(1_000)

    payroll = Payroll(bank_account)
    payroll.add_daily_pay('2021-01-01', c1)
    payroll.pay()

    print(bank_account.balance)

    payroll.add_daily_pay('2021-01-02', c1)
    payroll.pay()

    print(bank_account.balance)
```

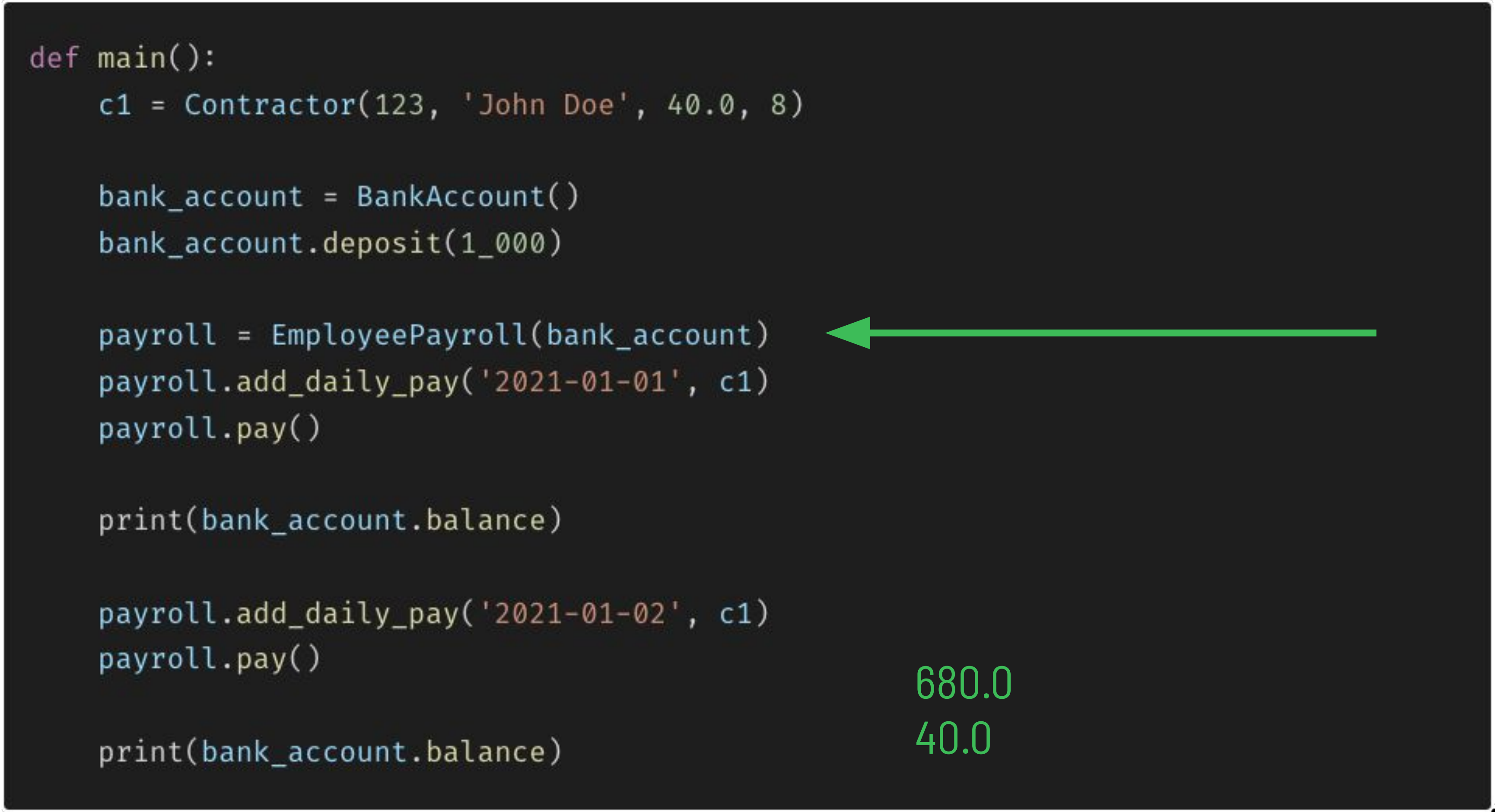
```
Send an email notifying this
Do other stuff
680.0
Send an email notifying this
Do other stuff
360.0
```



```python
class EmployeePayroll(Payroll):
    ...
    def pay(self):
        total = sum(entry.payment for entry in self._daily_payments_entries)
        if total > self.bank_account.balance:
            raise ValueError('Not enough funds')

        self.bank_account.withdraw(total)
```

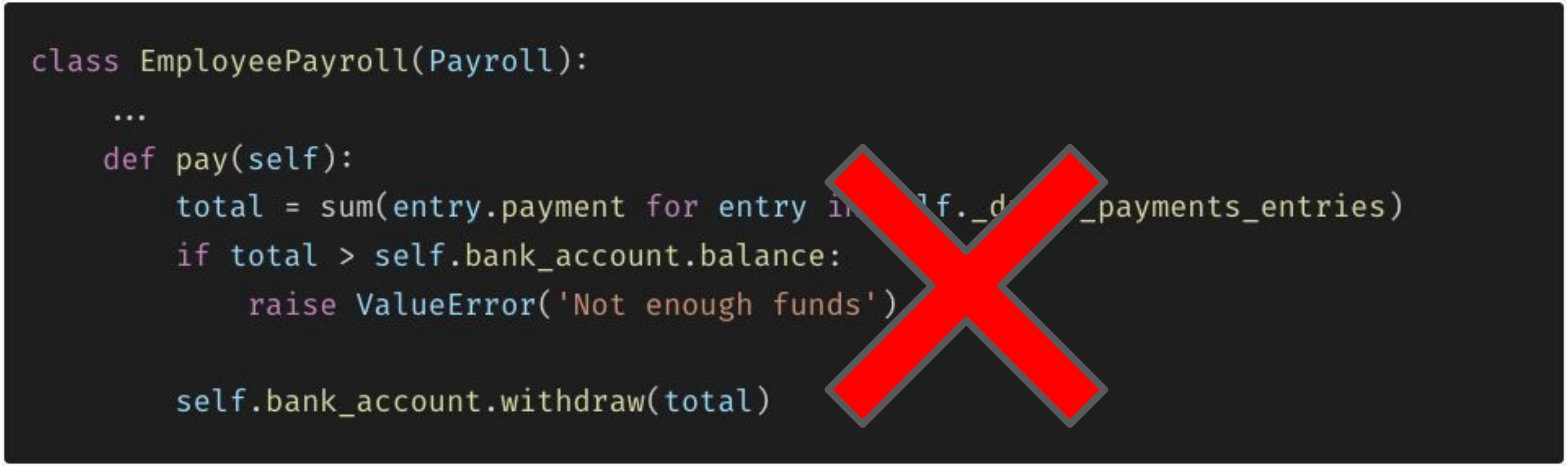


```python
def main():
    c1 = Contractor(123, 'John Doe', 40.0, 8)

    bank_account = BankAccount()
    bank_account.deposit(1_000)

    payroll = EmployeePayroll(bank_account)
    payroll.add_daily_pay('2021-01-01', c1)
    payroll.pay()

    print(bank_account.balance)

    payroll.add_daily_pay('2021-01-02', c1)
    payroll.pay()

    print(bank_account.balance)
```



```python
def main():
    c1 = Contractor(123, 'John Doe', 40.0, 8)

    bank_account = BankAccount()
    bank_account.deposit(1_000)

    payroll = EmployeePayroll(bank_account)
    payroll.add_daily_pay('2021-01-01', c1)
    payroll.pay()

    print(bank_account.balance)

    payroll.add_daily_pay('2021-01-02', c1)
    payroll.pay()

    print(bank_account.balance)
```

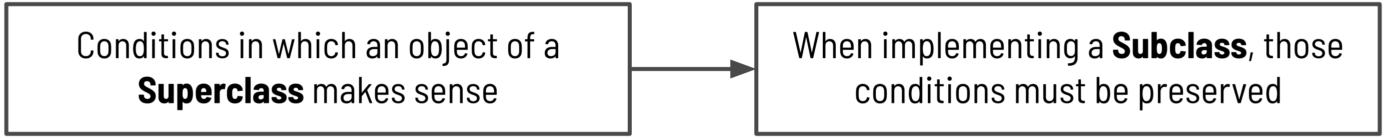
```
680.0
40.0
```



```
class EmployeePayroll(Payroll):
    ...
    def pay(self):
        total = sum(entry.payment for entry in self._debt_payments_entries)
        if total > self.bank_account.balance:
            raise ValueError('Not enough funds')

        self.bank_account.withdraw(total)
```

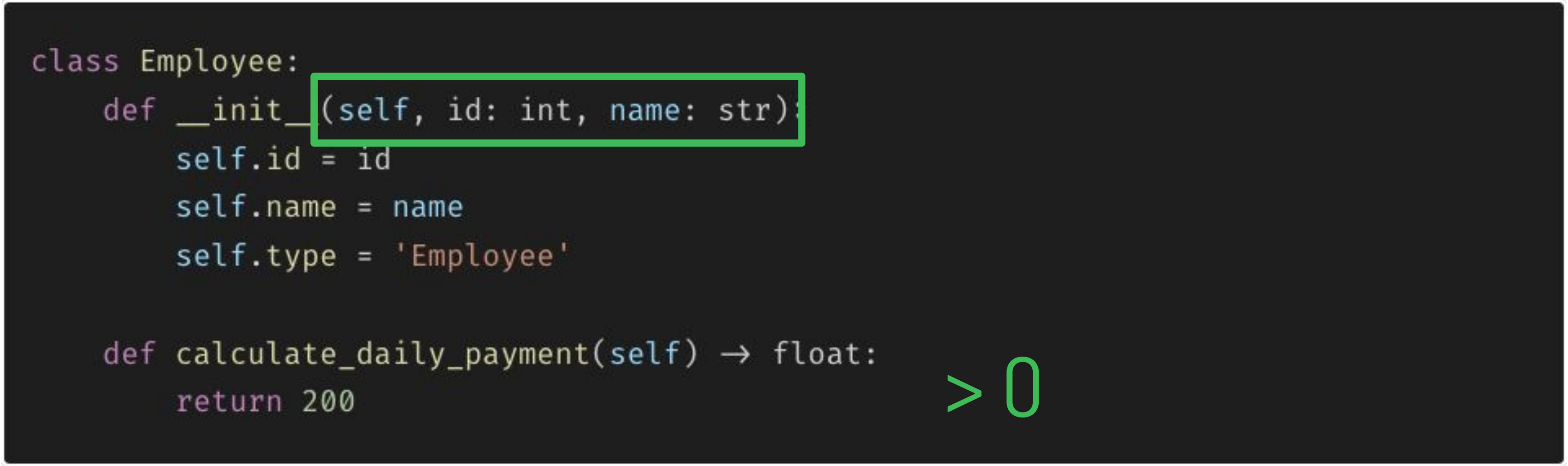
Check 6

The **invariants of a superclass** must be maintained or **preserved by its subclasses.**



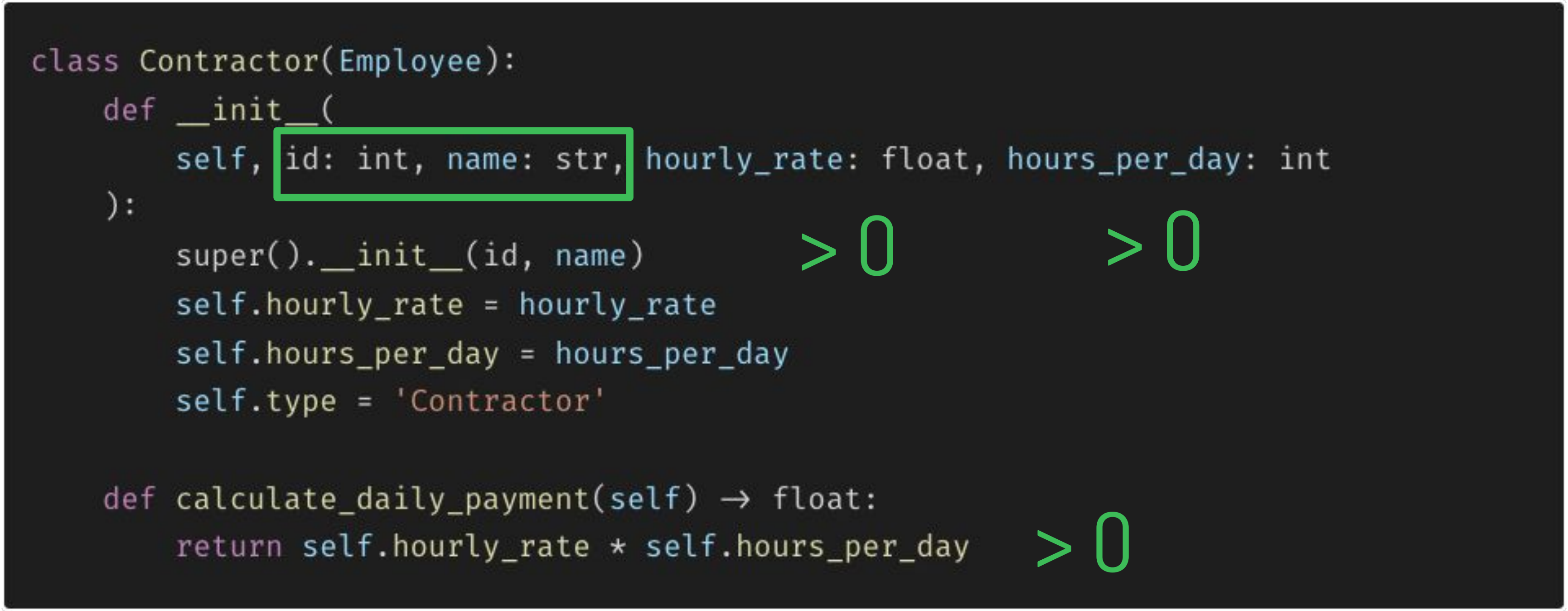
Conditions in which an object of a **Superclass** makes sense

When implementing a **Subclass**, those conditions must be preserved



```python
class Employee:
    def __init__(self, id: int, name: str):
        self.id = id
        self.name = name
        self.type = 'Employee'

    def calculate_daily_payment(self) -> float:
        return 200
```

> 0



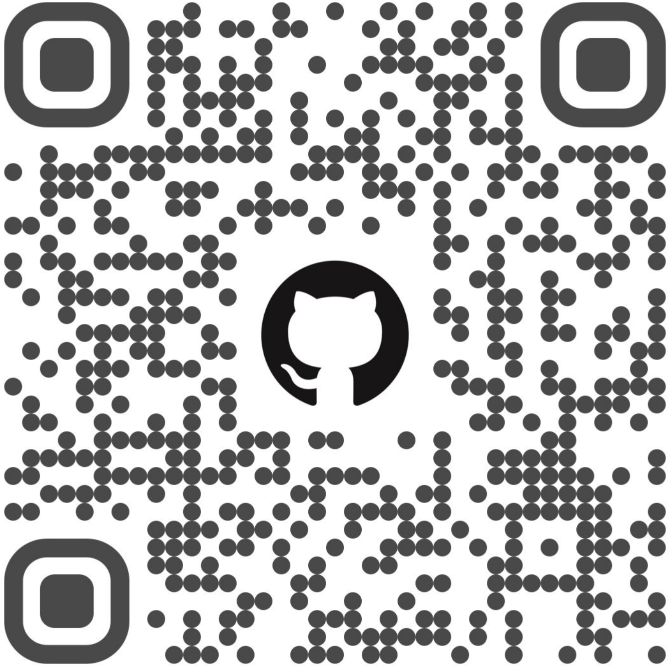
```python
class Contractor(Employee):
    def __init__(
        self, id: int, name: str, hourly_rate: float, hours_per_day: int
    ):
        super().__init__(id, name)
        self.hourly_rate = hourly_rate
        self.hours_per_day = hours_per_day
        self.type = 'Contractor'

    def calculate_daily_payment(self) → float:
        return self.hourly_rate * self.hours_per_day
```

> 0

> 0

> 0

Code



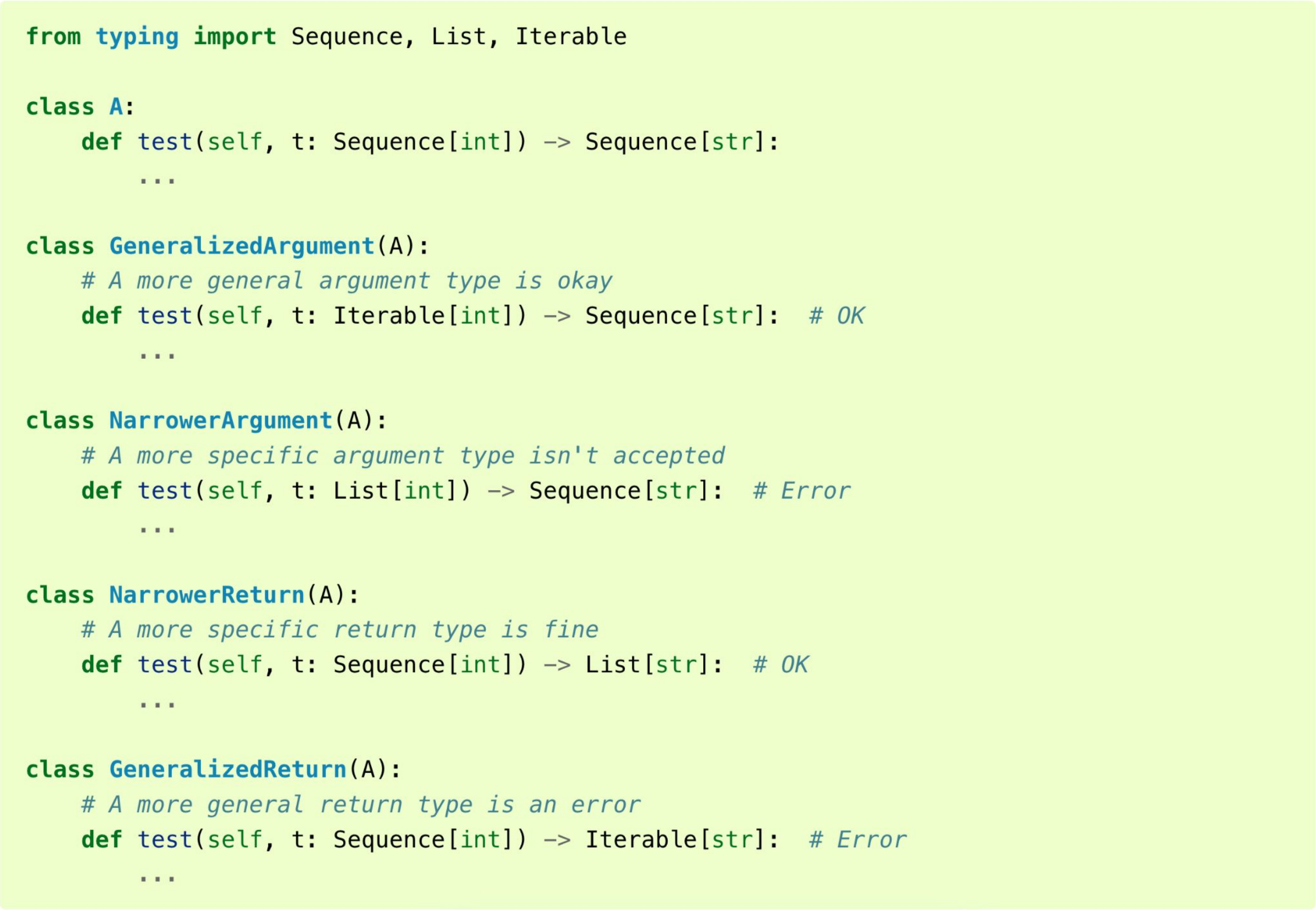
05

Using Mypy to check the LSP

Using Mypy to check the LSP

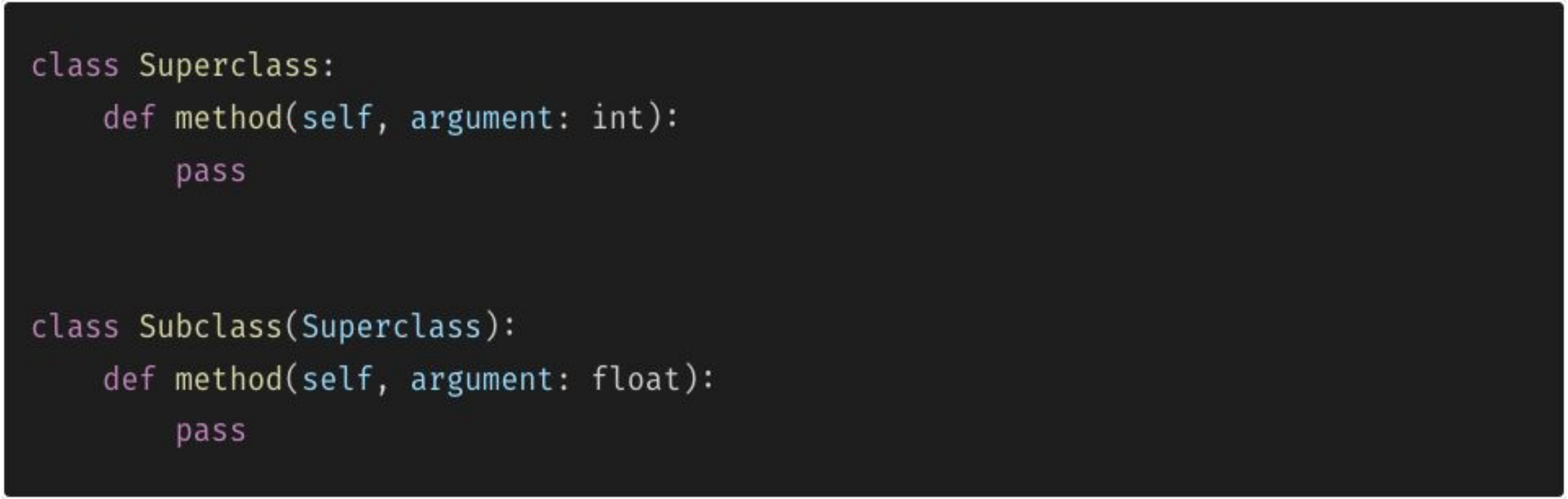
https://mypy.readthedocs.io/en/stable/common_issues.html#incompatible-overrides

https://mypy.readthedocs.io/en/stable/error_code_list.html#check-validity-of-overrides-override

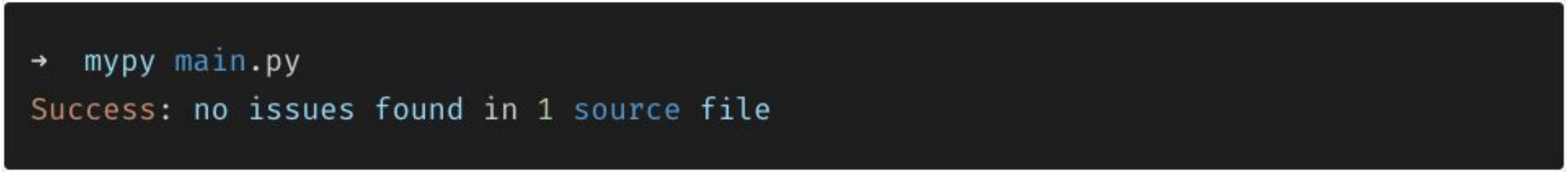


```python
from typing import Sequence, List, Iterable

class A:
    def test(self, t: Sequence[int]) -> Sequence[str]:
        ...

class GeneralizedArgument(A):
    # A more general argument type is okay
    def test(self, t: Iterable[int]) -> Sequence[str]:  # OK
        ...

class NarrowerArgument(A):
    # A more specific argument type isn't accepted
    def test(self, t: List[int]) -> Sequence[str]:  # Error
        ...

class NarrowerReturn(A):
    # A more specific return type is fine
    def test(self, t: Sequence[int]) -> List[str]:  # OK
        ...

class GeneralizedReturn(A):
    # A more general return type is an error
    def test(self, t: Sequence[int]) -> Iterable[str]:  # Error
        ...
```

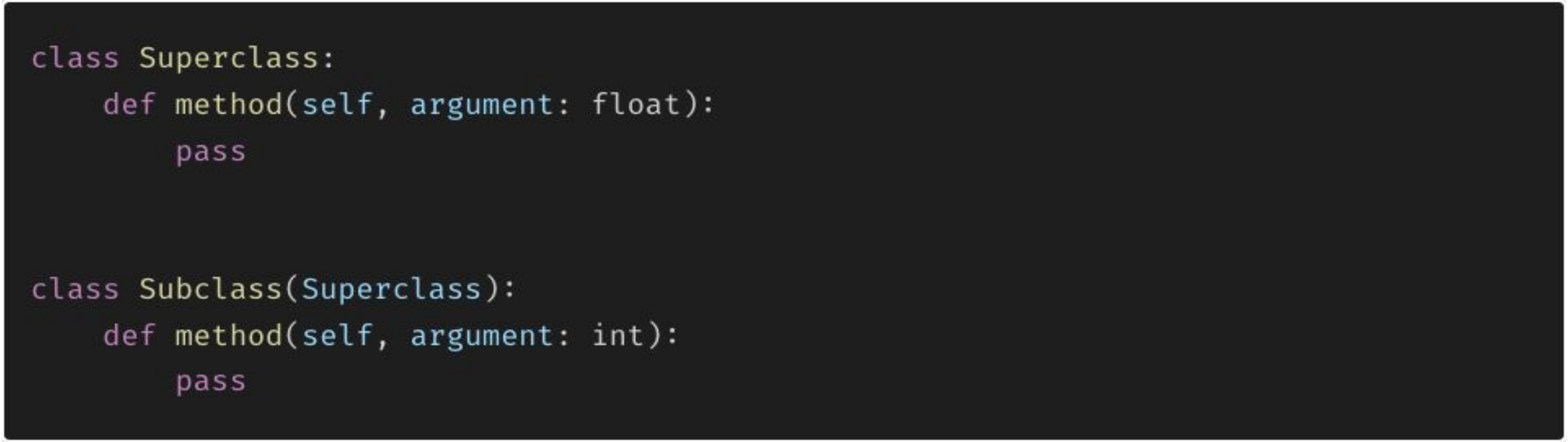
Using Mypy to check the LSP



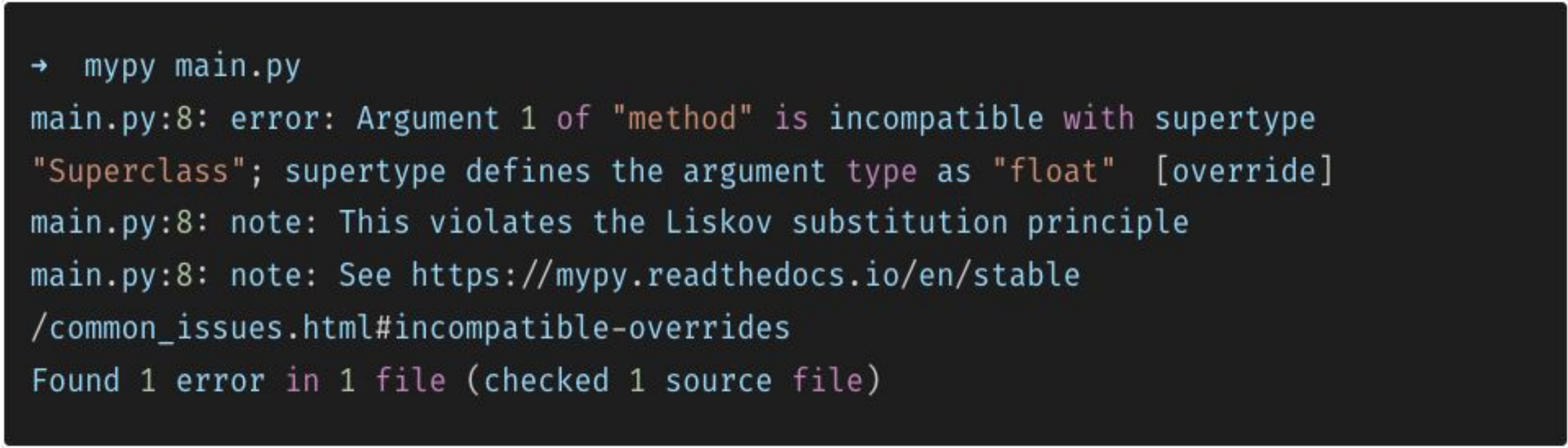
```python
class Superclass:
    def method(self, argument: int):
        pass


class Subclass(Superclass):
    def method(self, argument: float):
        pass
```



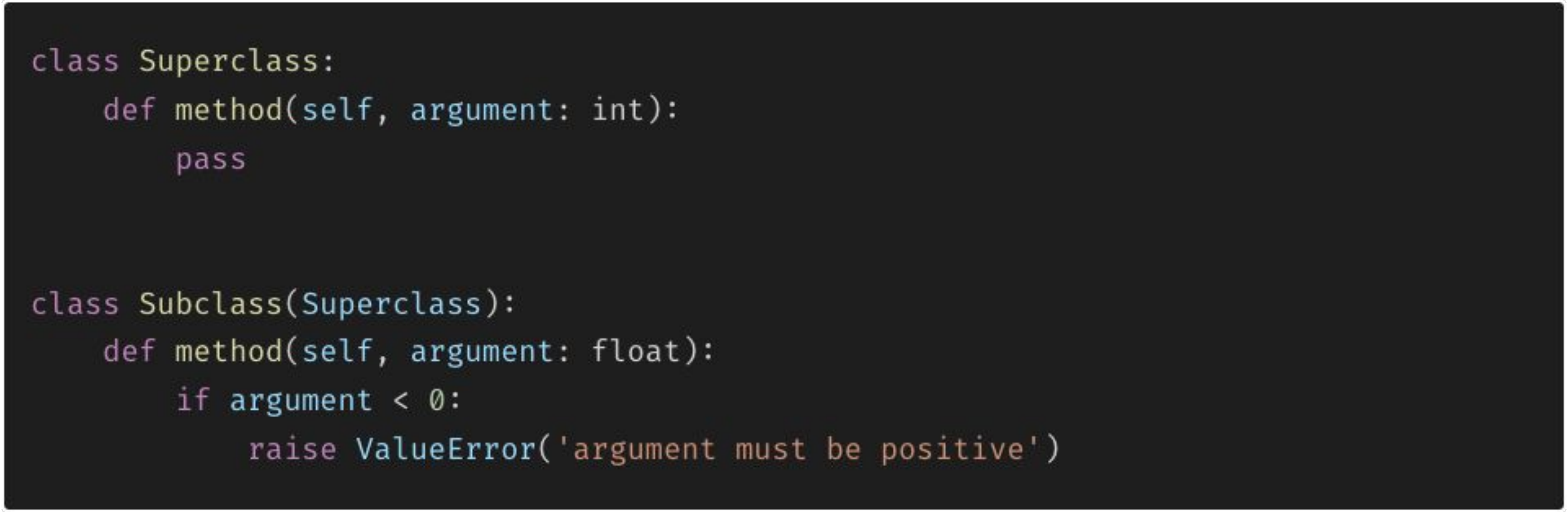
```
➜  mypy main.py
Success: no issues found in 1 source file
```



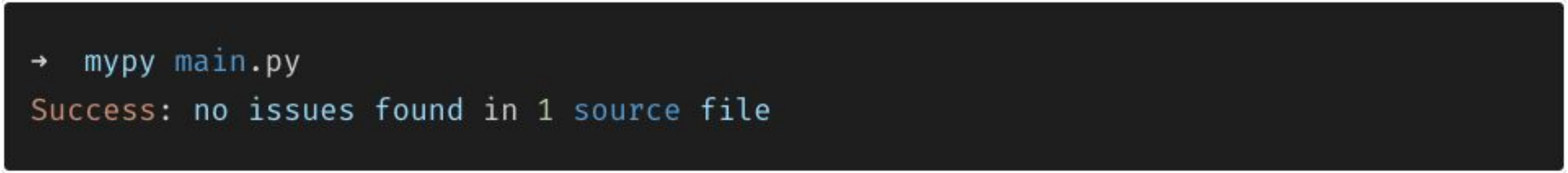
```python
class Superclass:
    def method(self, argument: float):
        pass


class Subclass(Superclass):
    def method(self, argument: int):
        pass
```



```
➜  mypy main.py
main.py:8: error: Argument 1 of "method" is incompatible with supertype
"Superclass"; supertype defines the argument type as "float"  [override]
main.py:8: note: This violates the Liskov substitution principle
main.py:8: note: See https://mypy.readthedocs.io/en/stable
/common_issues.html#incompatible-overrides
Found 1 error in 1 file (checked 1 source file)
```



```python
class Superclass:
    def method(self, argument: int):
        pass


class Subclass(Superclass):
    def method(self, argument: float):
        if argument < 0:
            raise ValueError('argument must be positive')
```



```
➜  mypy main.py
Success: no issues found in 1 source file
```

06

Benefits and Consequences of Violating LSP

Benefits

- Substitutability
- Polymorphism
- Modularity and extensibility

Consequences of Violating LSP

Unexpected behaviors

Fragile code

Reduced code reusability

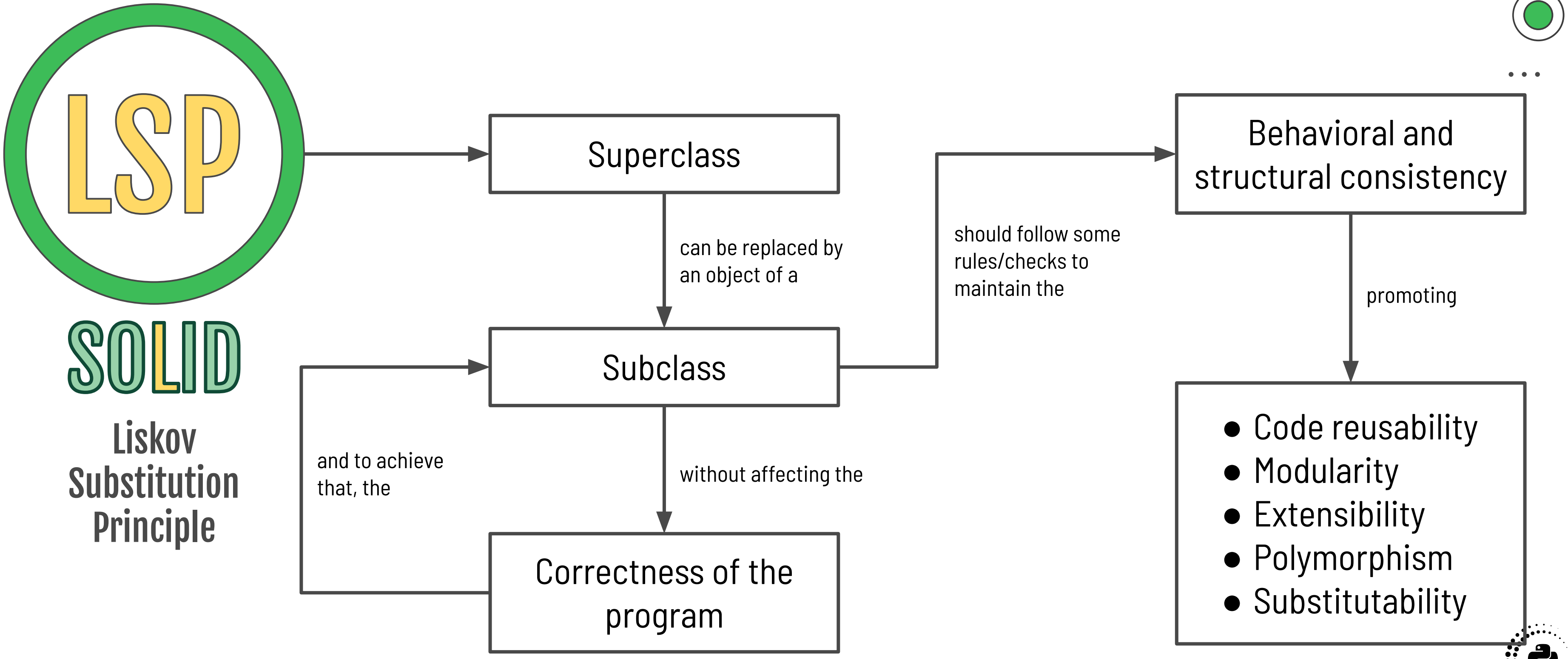
Maintenance challenges

Decreased extensibility

07

Summary and Conclusion

Summary



LSP

SOLID

Liskov Substitution Principle

Superclass

can be replaced by an object of a

Subclass

without affecting the

Correctness of the program

and to achieve that, the

should follow some rules/checks to maintain the

Behavioral and structural consistency

promoting

- Code reusability
- Modularity
- Extensibility
- Polymorphism
- Substitutability

Summary

Superclass	Subclass
Method parameters (arguments)	Match or be more general (or abstract)
Method return	Match or be a subtype
Exceptions	Match or be subtypes
Pre-conditions	Not to impose stricter preconditions
Post-conditions	Not to relax the post-conditions
Invariants	Must be preserved

Conclusion

Applying the LSP requires careful consideration of the contracts, preconditions, postconditions, and invariants defined by the superclass, as well as maintaining consistency in behavior and structure across the inheritance hierarchy.

By following the LSP, software developers can create robust, flexible, and maintainable code that stands the test of time. LSP forms a solid foundation for designing high-quality object-oriented systems.

Thanks!

Do you have any questions?

juanda@hourly.io
jdalzatec@gmail.com

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, infographics & images by Freepik, illustrations by Stories, and Business illustrations by Storyset