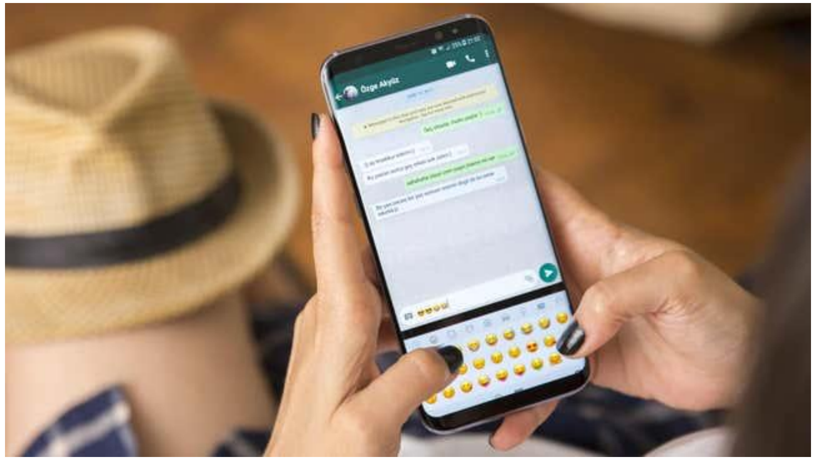
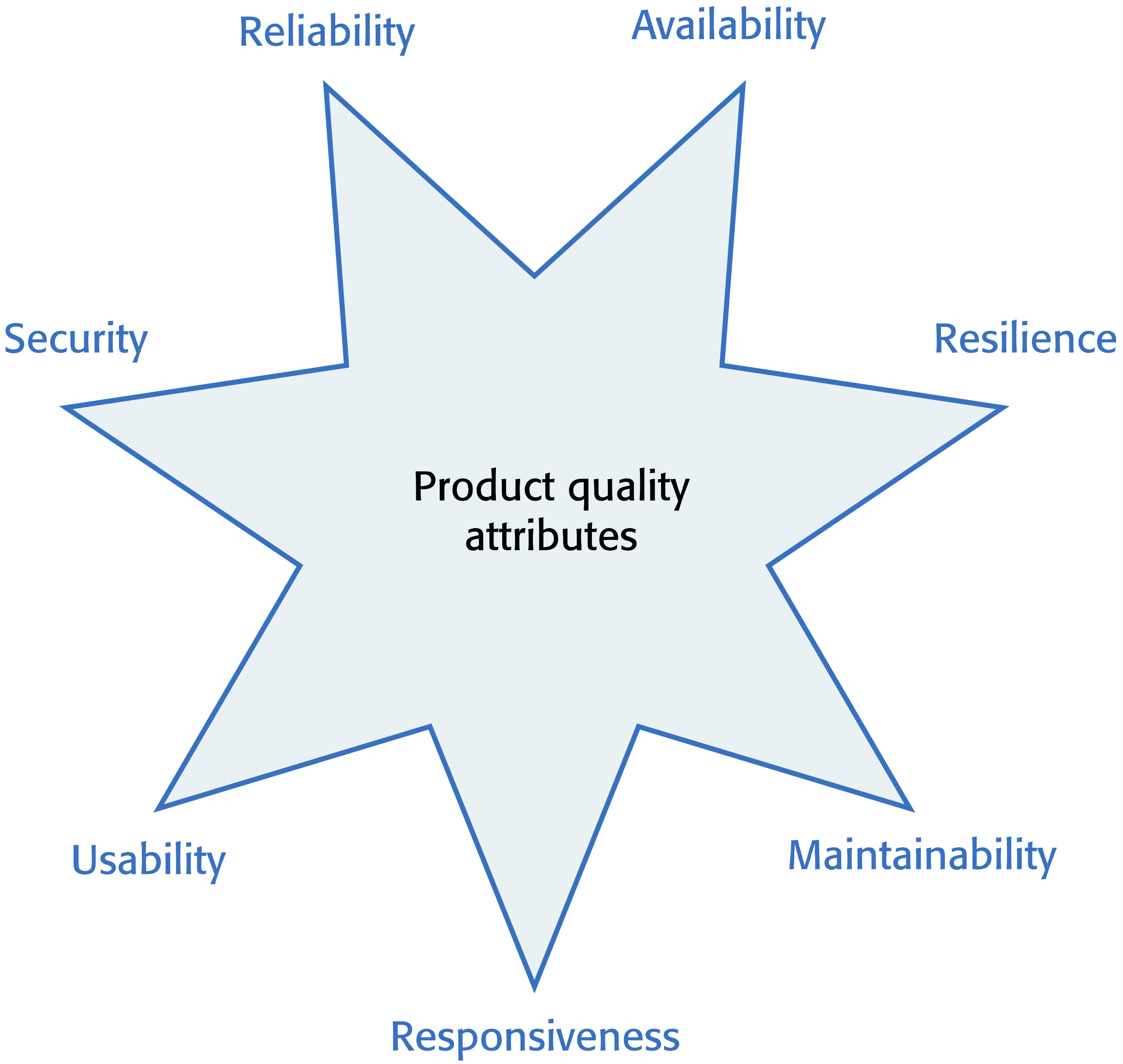
‘Reliable’ Programming

Examples from Sommerville’s 2018 book: “Engineering Software Products,
An Introduction to Modern Software Engineering”

Software Quality: Reliability



- Reliability is similar to trust
- Users want to know that software will perform consistently each time they use it.
  - If WhatsApp only sent 50% of our texts, that wouldn’t be very useful…
  - Likewise, we want Amazon to store keep our data safe…
  - We certainly want aeroplanes to take off and land safely 100% of the time!



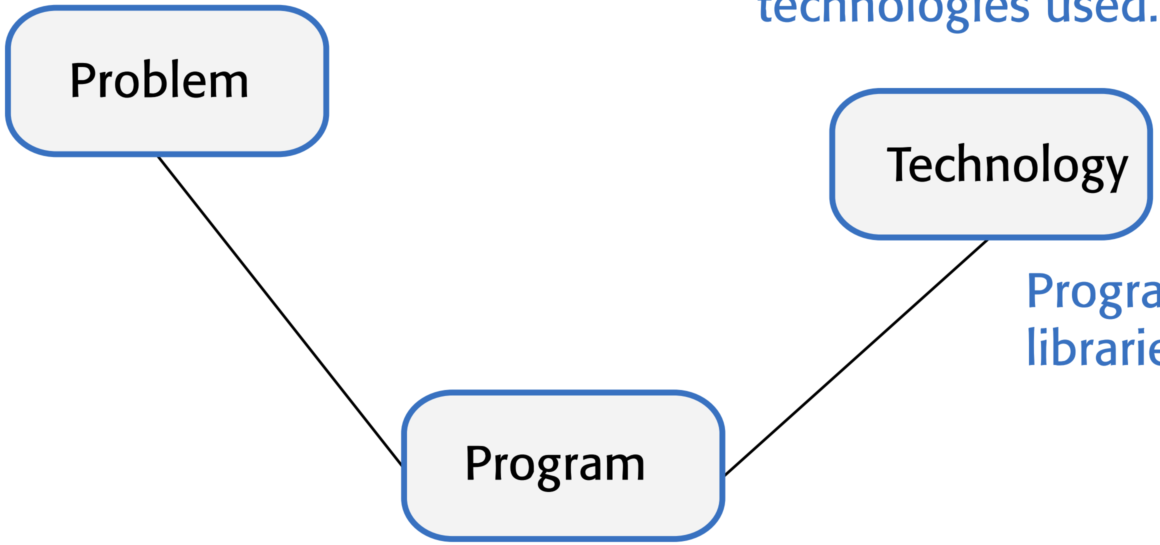
Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering

Three ways to improve reliability

- **Fault avoidance** - You should program in such a way that you avoid introducing faults into your program.
- **Input validation** - You should define the expected format for user inputs and validate that all inputs conform to that format.
- **Failure management** - You should implement your software so that program failures have minimal impact on product users.

Programmers make mistakes because they don’t properly understand the problem or the application domain.

Programmers make mistakes because they use unsuitable technology or they don’t properly understand the technologies used.



Programming language, libraries, database, IDE, etc.

Programmers make mistakes because they make simple slips or they do not completely understand how multiple program components work together and change the program’s state.

Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering

Single Responsibility

- Classes should model **one** entity:
  - Student
  - Player
  - NOT Student_Player_and_Course
- Attributes of a class obviously store **one** value at a time
- Methods to perform **one** action:
  - print()
  - remove()
  - insert()
  - NOT print_and_remove_and_insert_new()

- Single responsibility encourages cohesion and reuse within programs

Types of Complexity: Structural

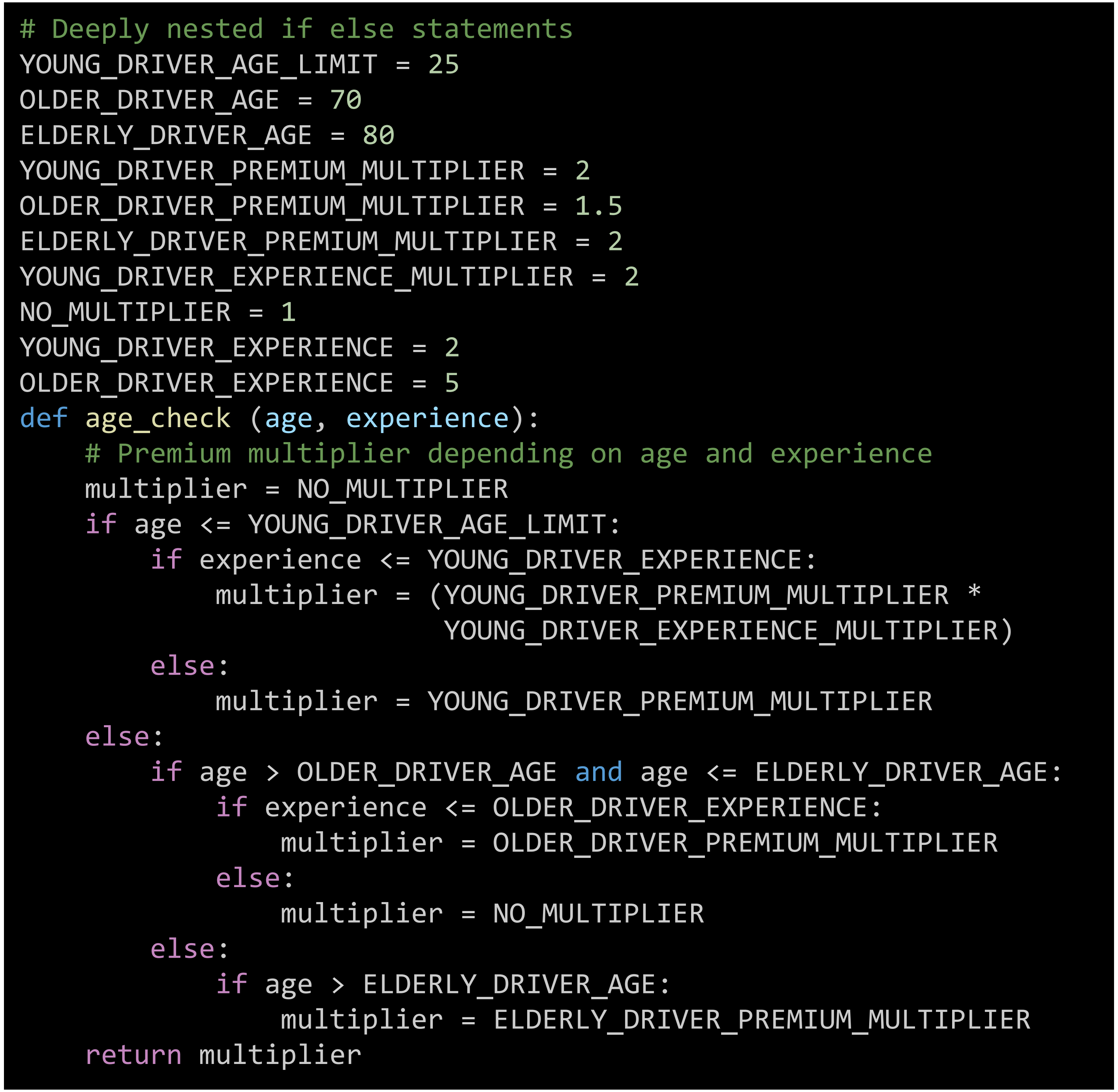
- **Structural complexity**
  - Functions should do one thing and one thing only
  - Functions should never have side-effects
  - Every class should have a single responsibility
  - Minimize the depth of inheritance hierarchies
  - Avoid multiple inheritance
  - Avoid threads (parallelism) unless absolutely necessary

Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering

Types of Complexity: Conditional

- **Conditional complexity**
  - Avoid deeply nested conditional statements
  - Avoid complex conditional expressions
- Deeply nested conditional (if) statements are used when you need to identify which of a possible set of choices is to be made.
- Consider the example code on the next slide

Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering

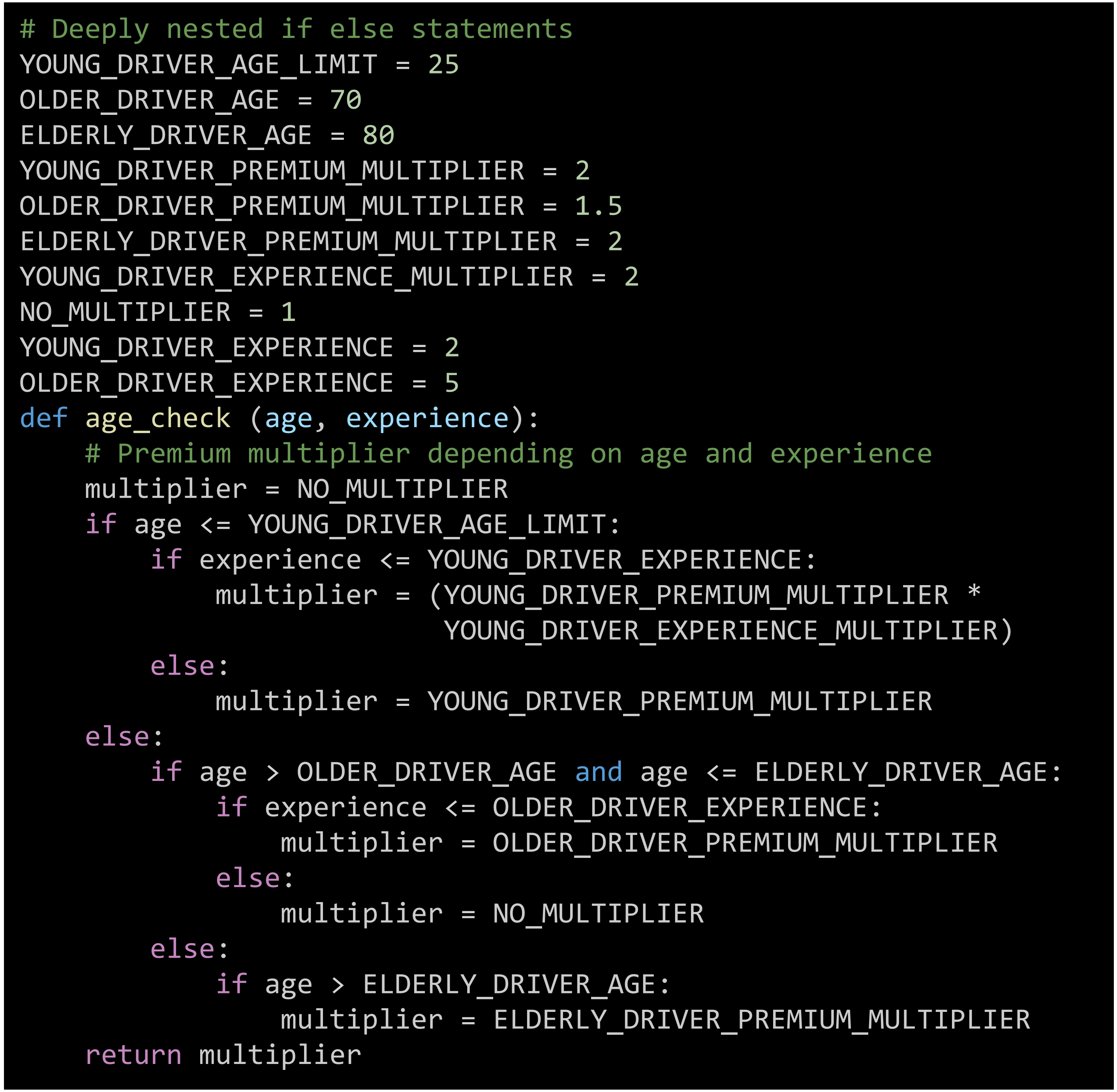


```
# Deeply nested if else statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    multiplier = NO_MULTIPLIER
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            multiplier = (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                          YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER
    else:
        if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
            if experience <= OLDER_DRIVER_EXPERIENCE:
                multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER
            else:
                multiplier = NO_MULTIPLIER
        else:
            if age > ELDERLY_DRIVER_AGE:
                multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER
    return multiplier
```

Example of Condition Complexity

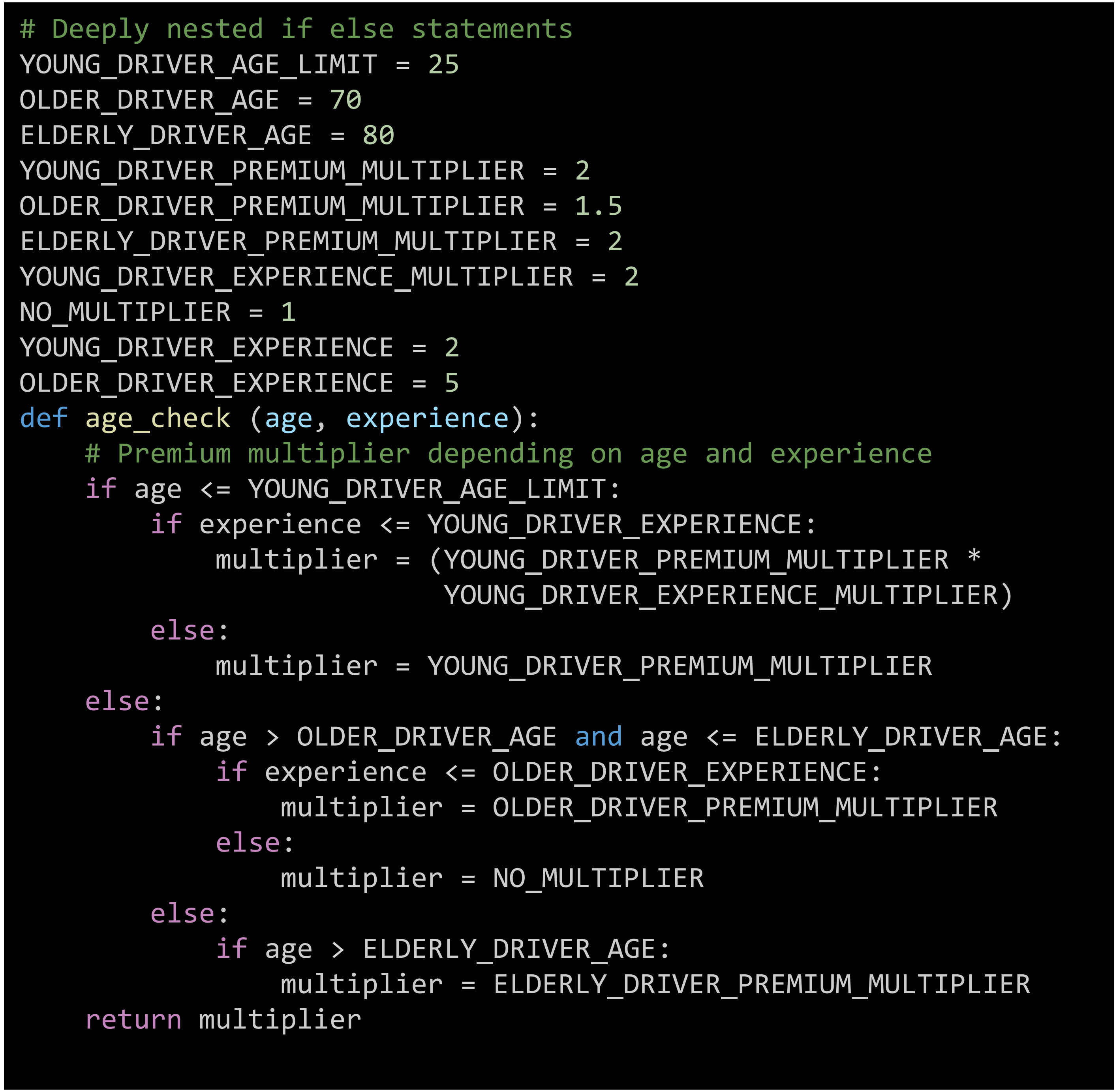
- Deeply nested conditional (if) statements are used when you need to identify which of a possible set of choices is to be made.
- For example, the function ‘age_check’ is a short Python function that is used to calculate an age multiplier for insurance premiums.
- The insurance company’s data suggests that the age and experience of drivers affects the chances of them having an accident, so premiums are adjusted to take this into account.
- It is good practice to name constants rather than using absolute numbers, so the program names all constants that are used.

Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering



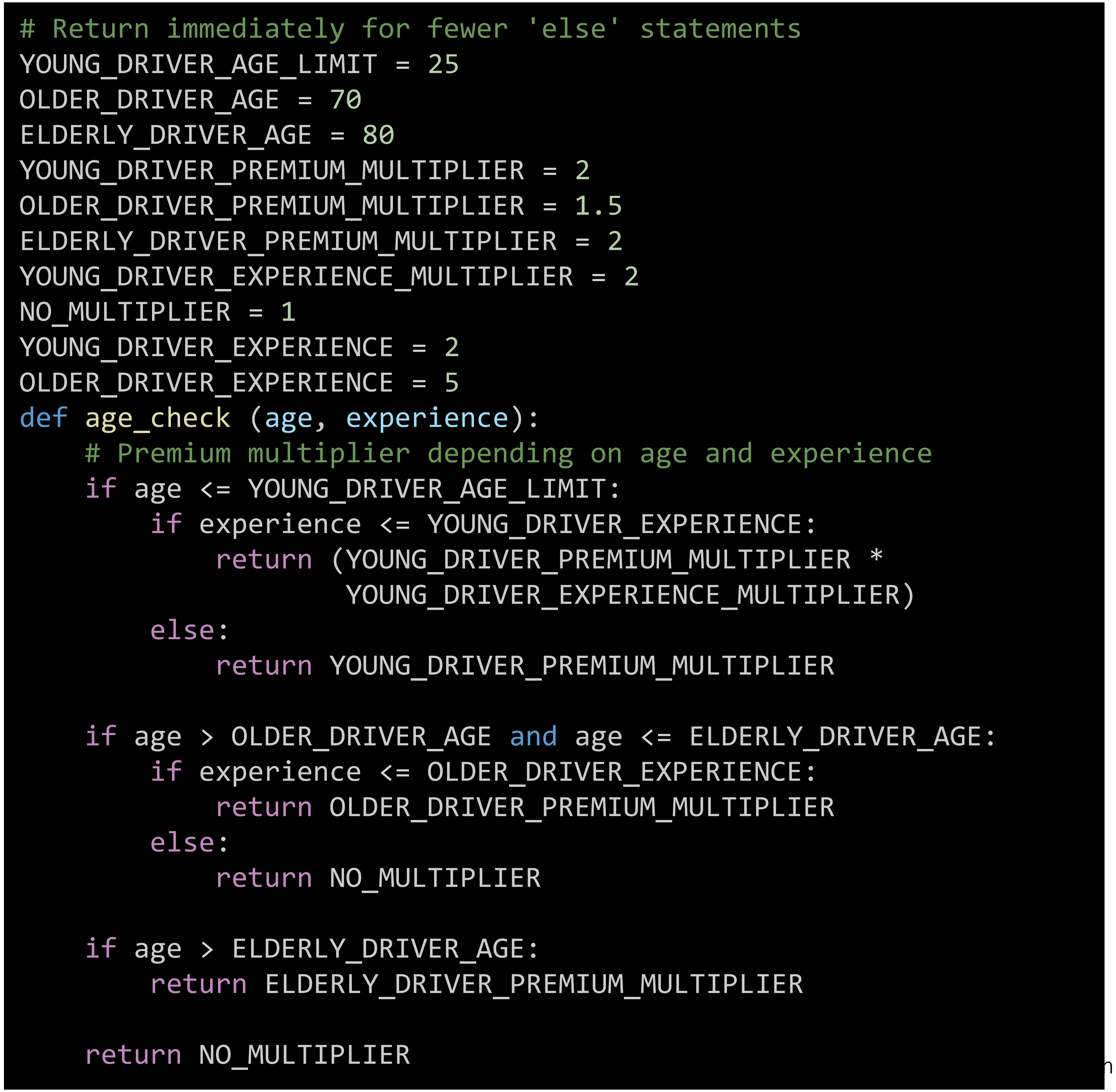
```python
# Deeply nested if else statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    multiplier = NO_MULTIPLIER
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            multiplier = (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                          YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER
    else:
        if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
            if experience <= OLDER_DRIVER_EXPERIENCE:
                multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER
            else:
                multiplier = NO_MULTIPLIER
        else:
            if age > ELDERLY_DRIVER_AGE:
                multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER
    return multiplier
```

merville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering



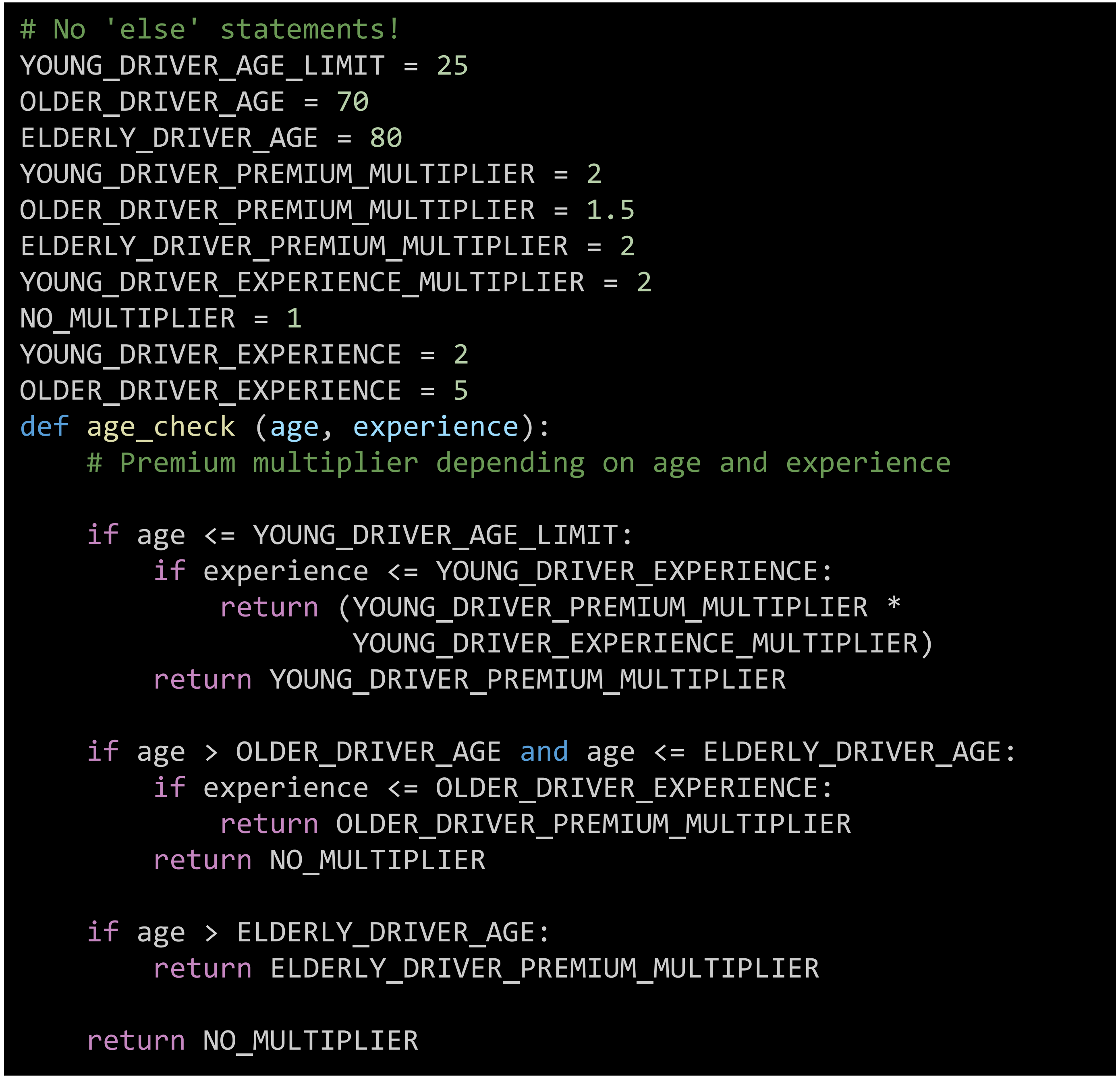
```
# Deeply nested if else statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            multiplier = (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                          YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            multiplier = YOUNG_DRIVER_PREMIUM_MULTIPLIER
    else:
        if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
            if experience <= OLDER_DRIVER_EXPERIENCE:
                multiplier = OLDER_DRIVER_PREMIUM_MULTIPLIER
            else:
                multiplier = NO_MULTIPLIER
        else:
            if age > ELDERLY_DRIVER_AGE:
                multiplier = ELDERLY_DRIVER_PREMIUM_MULTIPLIER
    return multiplier
```

merville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering



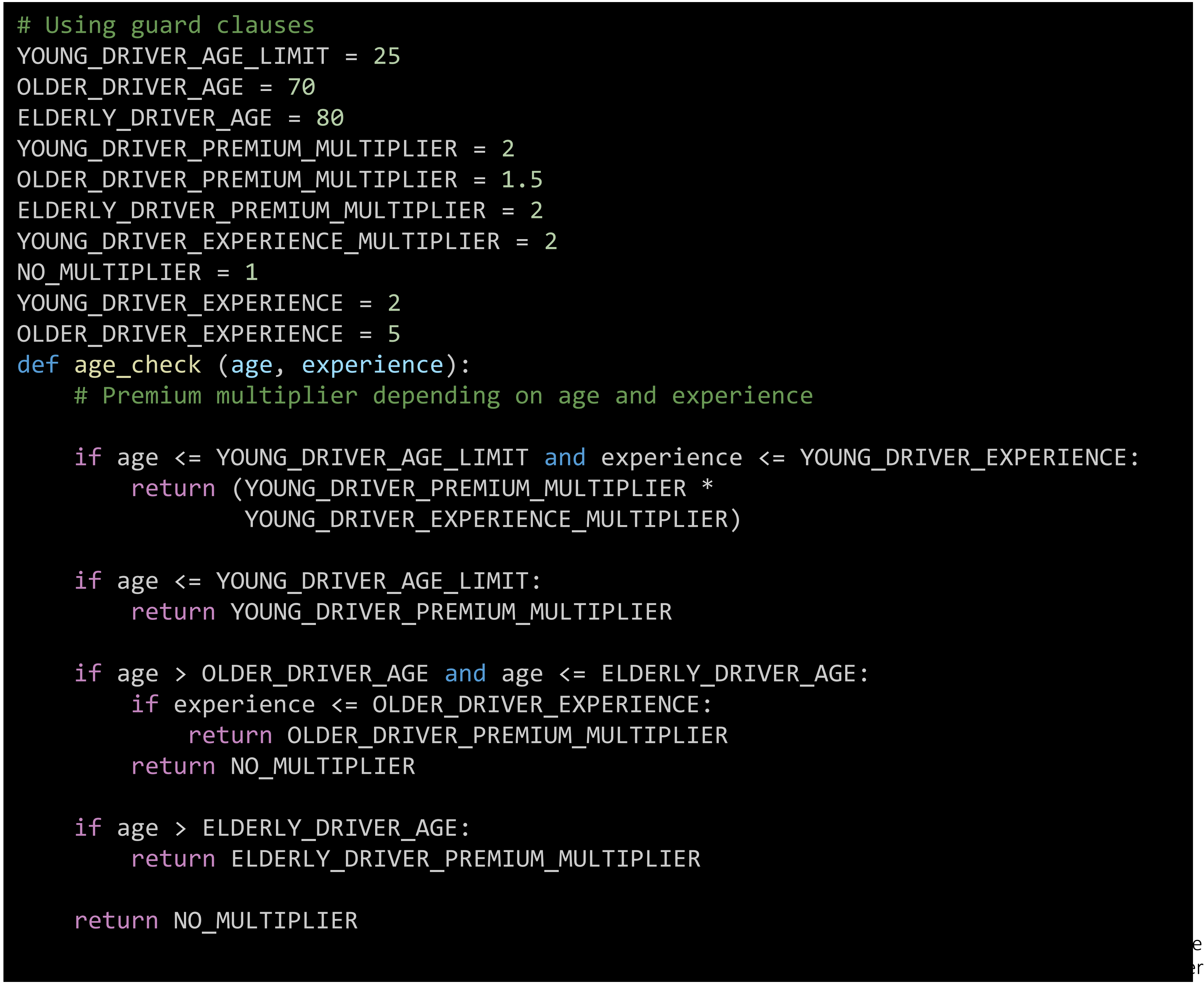
```python
# Return immediately for fewer 'else' statements
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience
    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                    YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        else:
            return YOUNG_DRIVER_PREMIUM_MULTIPLIER

    if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
        if experience <= OLDER_DRIVER_EXPERIENCE:
            return OLDER_DRIVER_PREMIUM_MULTIPLIER
        else:
            return NO_MULTIPLIER

    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

merville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering

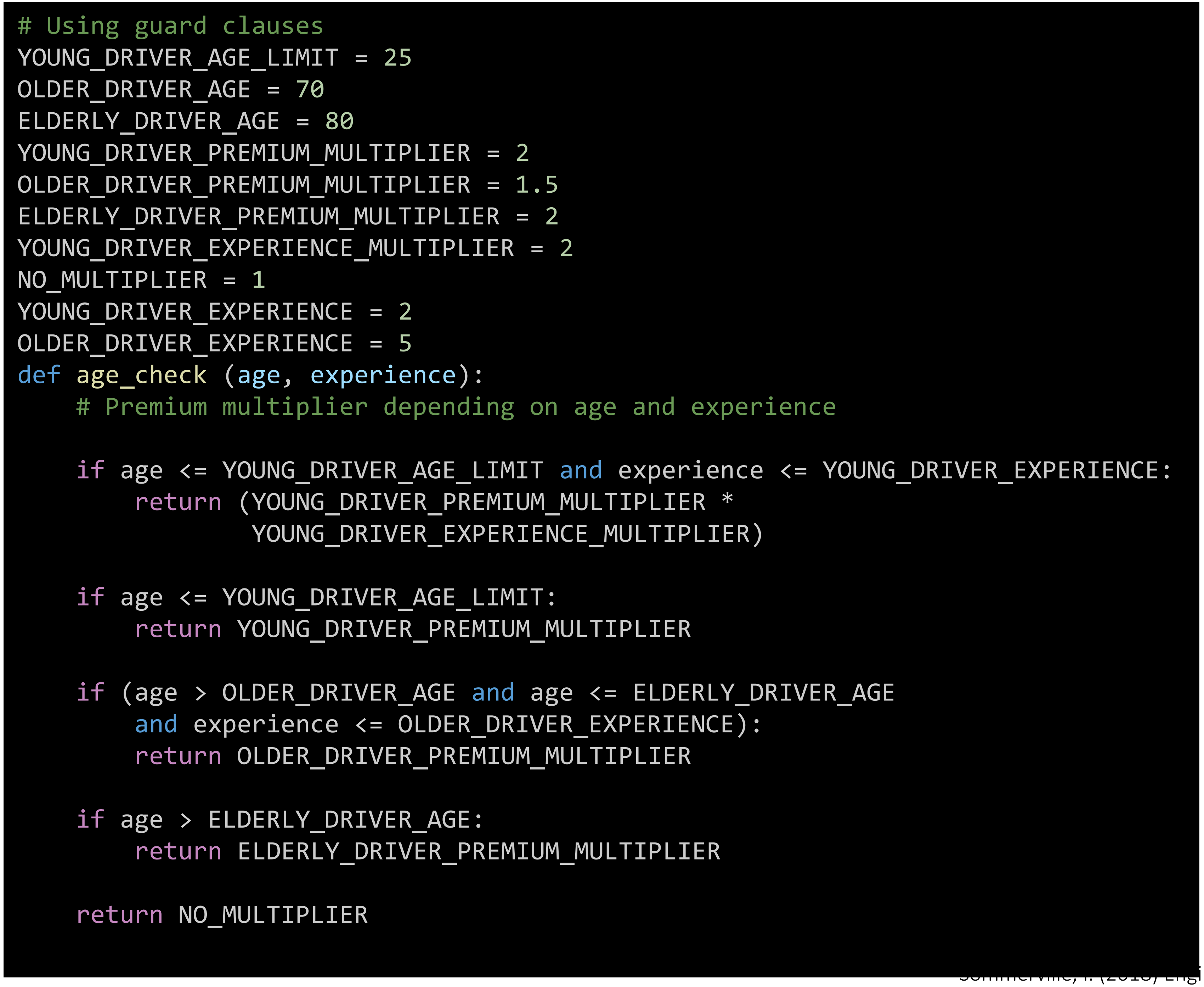


```python
# No 'else' statements!
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience

    if age <= YOUNG_DRIVER_AGE_LIMIT:
        if experience <= YOUNG_DRIVER_EXPERIENCE:
            return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                    YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER

    if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
        if experience <= OLDER_DRIVER_EXPERIENCE:
            return OLDER_DRIVER_PREMIUM_MULTIPLIER
        return NO_MULTIPLIER

    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering



```
# Using guard clauses
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience

    if age <= YOUNG_DRIVER_AGE_LIMIT and experience <= YOUNG_DRIVER_EXPERIENCE:
        return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)

    if age <= YOUNG_DRIVER_AGE_LIMIT:
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER

    if age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE:
        if experience <= OLDER_DRIVER_EXPERIENCE:
            return OLDER_DRIVER_PREMIUM_MULTIPLIER
        return NO_MULTIPLIER

    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

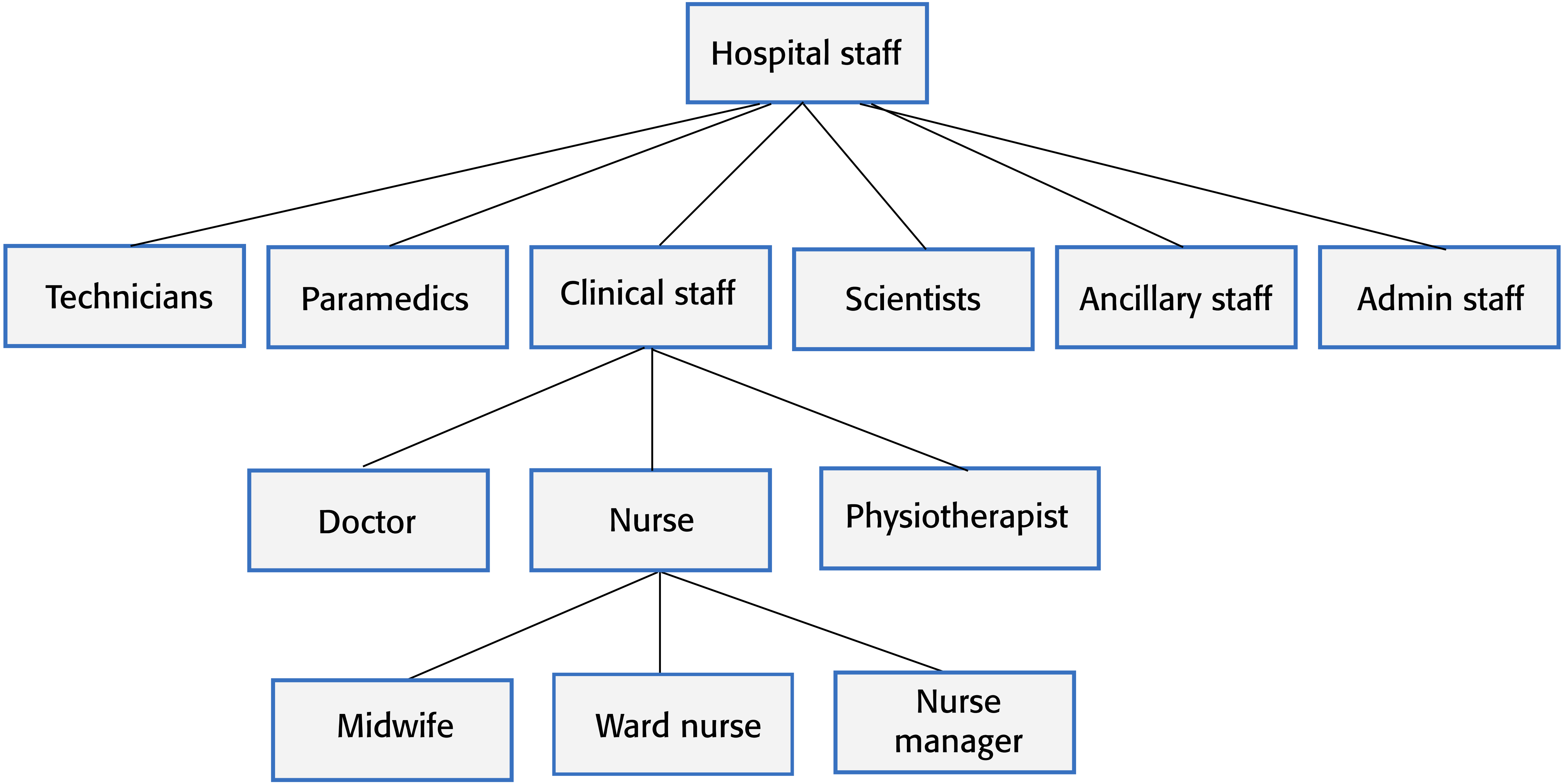


```python
# Using guard clauses
YOUNG_DRIVER_AGE_LIMIT = 25
OLDER_DRIVER_AGE = 70
ELDERLY_DRIVER_AGE = 80
YOUNG_DRIVER_PREMIUM_MULTIPLIER = 2
OLDER_DRIVER_PREMIUM_MULTIPLIER = 1.5
ELDERLY_DRIVER_PREMIUM_MULTIPLIER = 2
YOUNG_DRIVER_EXPERIENCE_MULTIPLIER = 2
NO_MULTIPLIER = 1
YOUNG_DRIVER_EXPERIENCE = 2
OLDER_DRIVER_EXPERIENCE = 5
def age_check (age, experience):
    # Premium multiplier depending on age and experience

    if age <= YOUNG_DRIVER_AGE_LIMIT and experience <= YOUNG_DRIVER_EXPERIENCE:
        return (YOUNG_DRIVER_PREMIUM_MULTIPLIER *
                YOUNG_DRIVER_EXPERIENCE_MULTIPLIER)

    if age <= YOUNG_DRIVER_AGE_LIMIT:
        return YOUNG_DRIVER_PREMIUM_MULTIPLIER

    if (age > OLDER_DRIVER_AGE and age <= ELDERLY_DRIVER_AGE
        and experience <= OLDER_DRIVER_EXPERIENCE):
        return OLDER_DRIVER_PREMIUM_MULTIPLIER

    if age > ELDERLY_DRIVER_AGE:
        return ELDERLY_DRIVER_PREMIUM_MULTIPLIER

    return NO_MULTIPLIER
```

Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering

Structural complexity: avoid deep inheritance

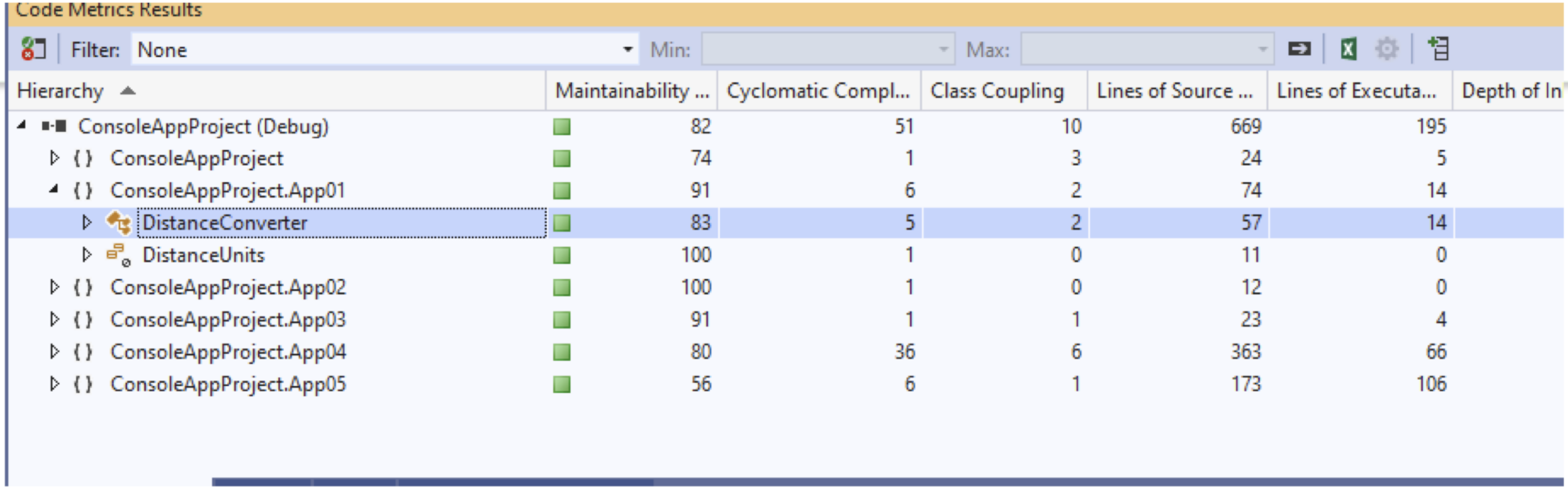
- Inheritance appears to be an effective and efficient way of reusing code and of making changes that affect all subclasses.
- However, inheritance increases the structural complexity of code as it increases the coupling of subclasses. The diagram shows part of a 4-level inheritance hierarchy that could be defined for staff in a hospital.


Hospital staff
Technicians
Paramedics
Clinical staff
Scientists
Ancillary staff
Admin staff
Doctor
Nurse
Physiotherapist
Midwife
Ward nurse
Nurse manager

Structural complexity: avoid deep inheritance

- The problem with deep inheritance is that if you want to make changes to a class, you have to look at all of its superclasses to see where it is best to make the change.
- You also have to look at all of the related subclasses to check that the change does not have unwanted consequences. It’s easy to make mistakes when you are doing this analysis and introduce faults into your program.

Measuring Quality: Code Metrics in VS22



Code Metrics Results

Filter: None Min: Max:

Hierarchy		Maintainability ...	Cyclomatic Compl...	Class Coupling	Lines of Source ...	Lines of Executa...	Depth of In
ConsoleAppProject (Debug)	🟩	82	51	10	669	195	
ConsoleAppProject	🟩	74	1	3	24	5	
ConsoleAppProject.App01	🟩	91	6	2	74	14	
DistanceConverter	🟩	83	5	2	57	14	
DistanceUnits	🟩	100	1	0	11	0	
ConsoleAppProject.App02	🟩	100	1	0	12	0	
ConsoleAppProject.App03	🟩	91	1	1	23	4	
ConsoleAppProject.App04	🟩	80	36	6	363	66	
ConsoleAppProject.App05	🟩	56	6	1	173	106	

Maintainability

- Green 20 -1 00
- Yellow 10 – 19
- Red 0 – 9
- (Higher the better)

Complexity

- Lower the better

Lines of Code

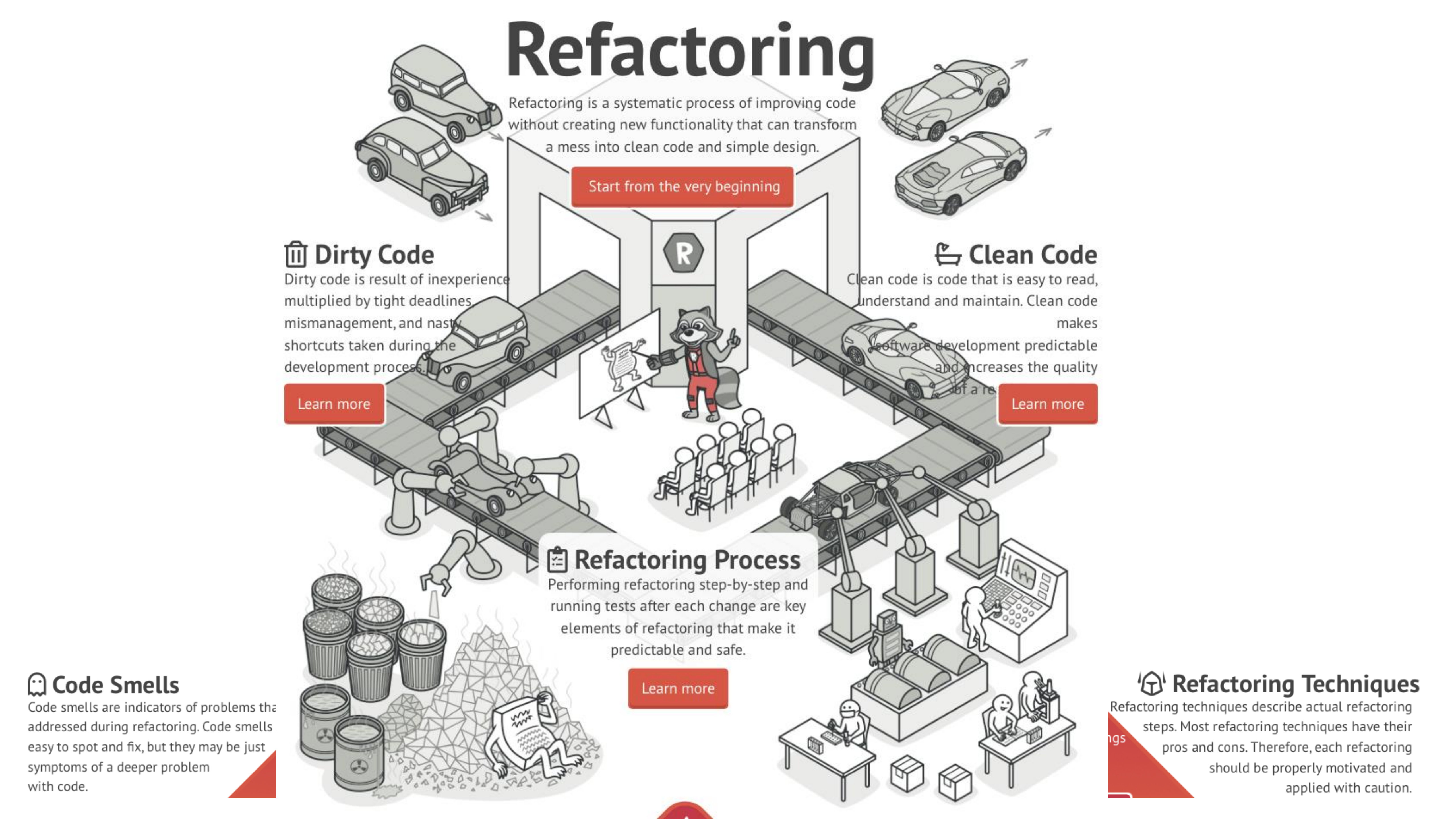
- Lower the better

Coupling

- Lower the better

Inheritance Depth

- Lower the better



Refactoring

Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design.

Dirty Code

Dirty code is result of inexperience multiplied by tight deadlines, mismanagement, and nasty shortcuts taken during the development process.

Clean Code

Clean code is code that is easy to read, understand and maintain. Clean code makes software development predictable and increases the quality of a re

Refactoring Process

Performing refactoring step-by-step and running tests after each change are key elements of refactoring that make it predictable and safe.

Code Smells

Code smells are indicators of problems tha addressed during refactoring. Code smells easy to spot and fix, but they may be just symptoms of a deeper problem with code.

Refactoring Techniques

Refactoring techniques describe actual refactoring steps. Most refactoring techniques have their pros and cons. Therefore, each refactoring should be properly motivated and applied with caution.

Code ‘smells’

- **‘Code smells’ are indicators in the code that there might be a deeper problem.**
- Martin Fowler, a refactoring pioneer, suggests that the starting point for refactoring should be to identify code smells.
- For example, very large classes may indicate that the class is trying to do too much. This probably means that its structural complexity is high.

Martin Fowler

Examples of Code ‘smells’

Large classes

- Large classes may mean that the single responsibility principle is being violated.
- Break down large classes into easier-to-understand, smaller classes.

Long methods/functions

- Long methods or functions may indicate that the function is doing more than one thing.
- Split into smaller, more specific functions or methods.

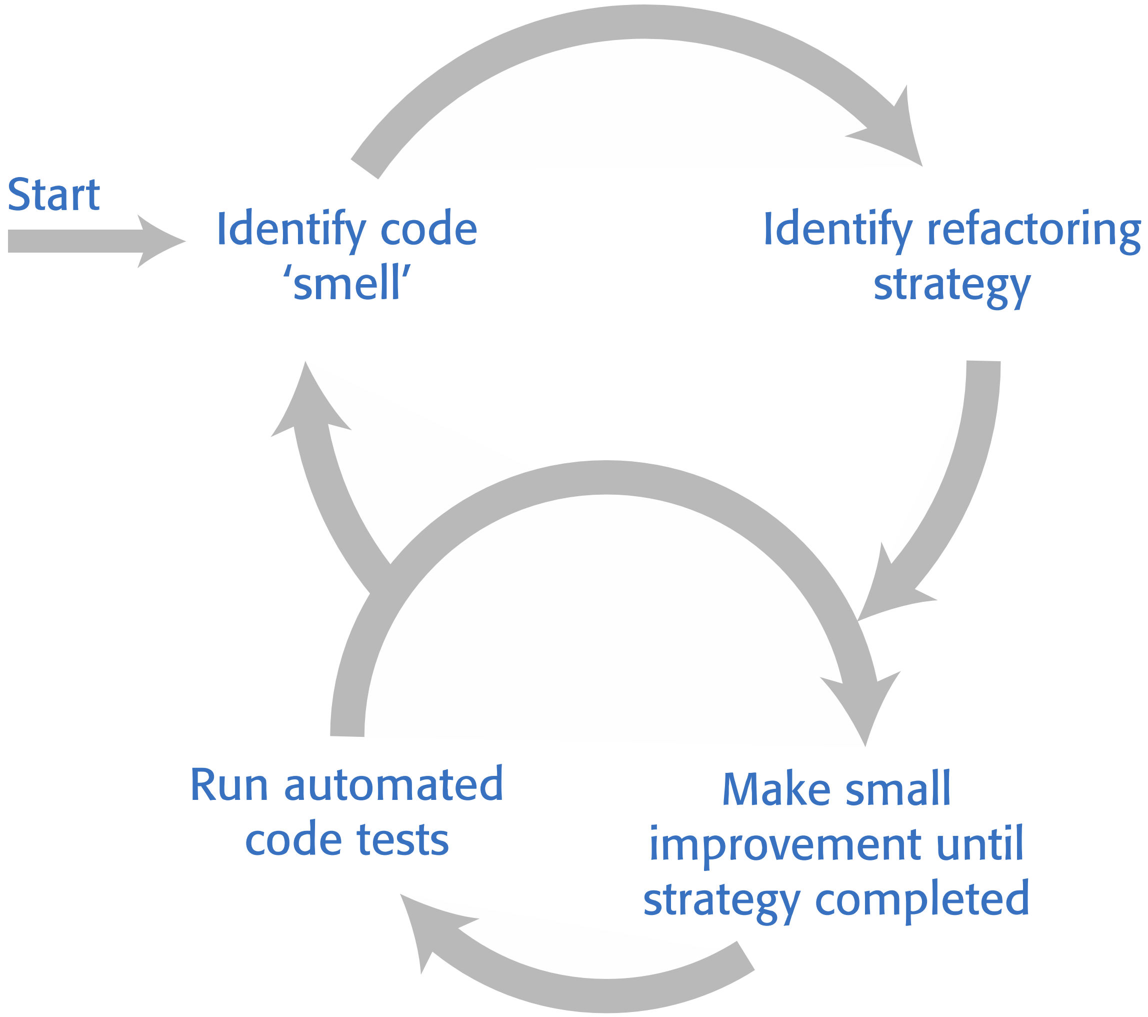
Duplicated code

- Duplicated code may mean that when changes are needed, these have to be made everywhere the code is duplicated.
- Rewrite to create a single instance of the duplicated code that is used as required

Meaningless names

- Meaningless names are a sign of programmer haste. They make the code harder to understand.
- Replace with meaningful names and check for other shortcuts that the programmer may have taken.

Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering

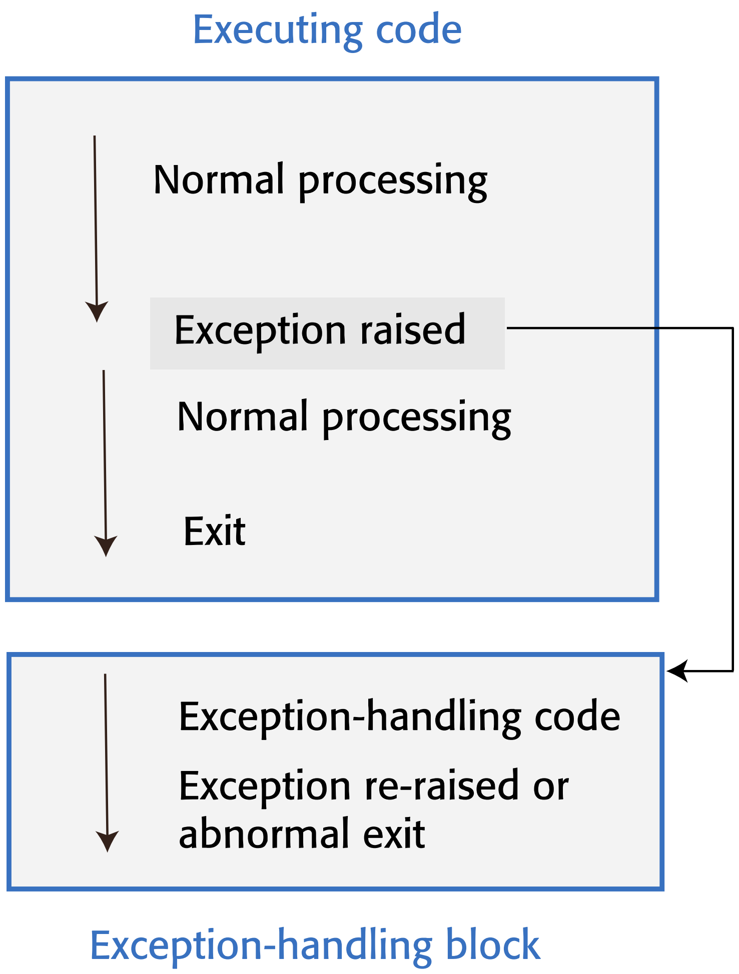


Start
Identify code 'smell'
Identify refactoring strategy
Make small improvement until strategy completed
Run automated code tests

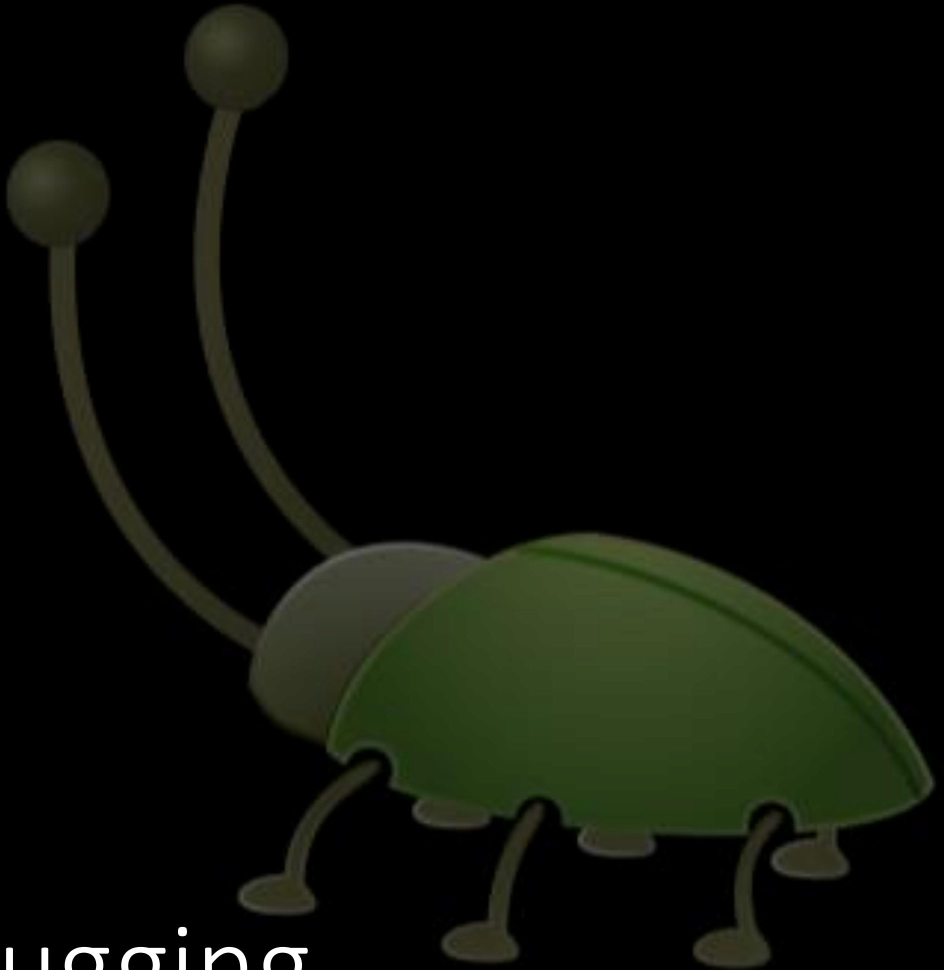
- Refactoring means changing a program to reduce its complexity without changing the external behaviour of that program.

Exception Handling

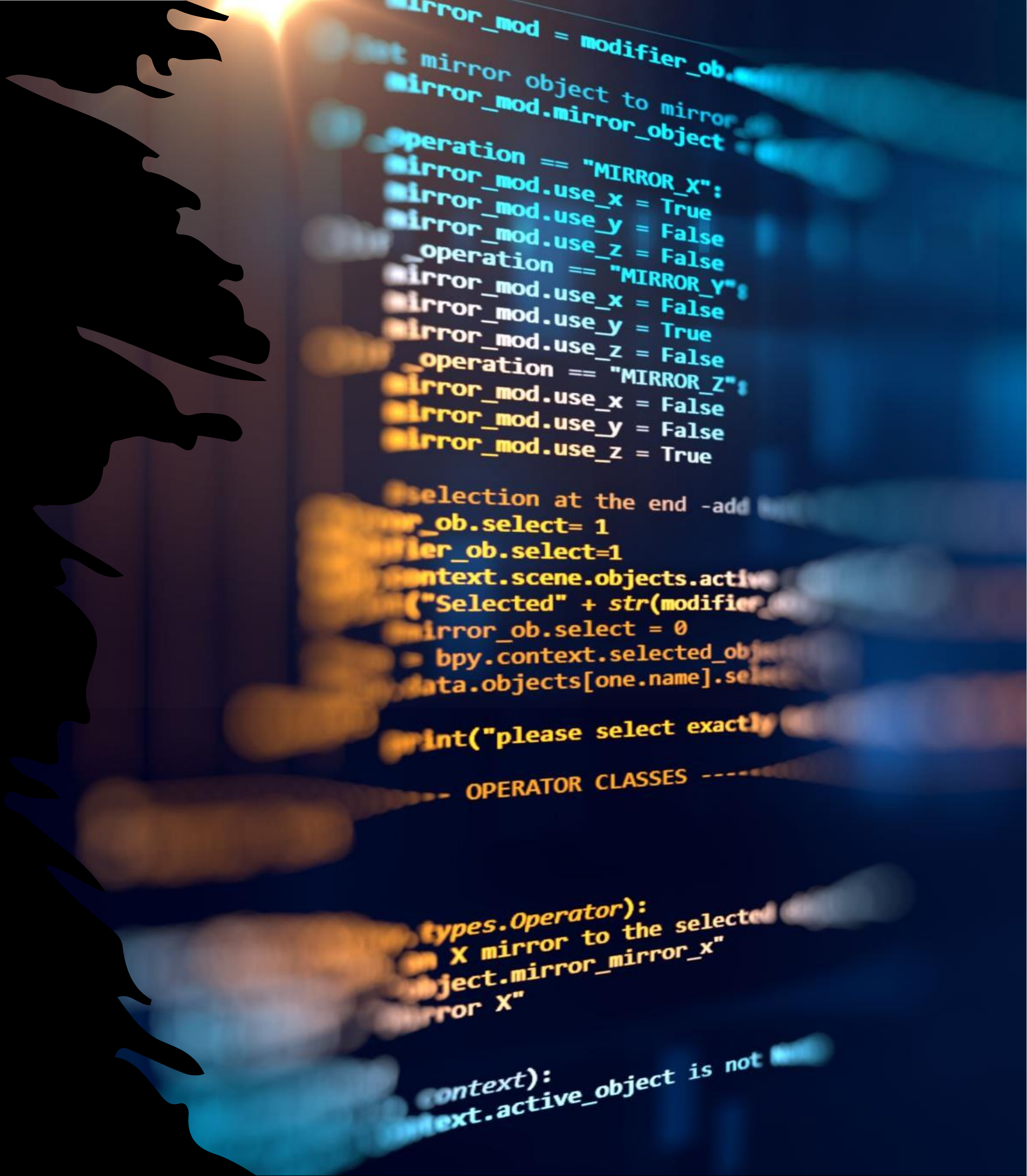
- Exceptions are events that disrupt the normal flow of processing in a program.
- In Python, you use **`try-except`** keywords to indicate exception handling code; in Java, the equivalent keywords are **`try-catch`**.



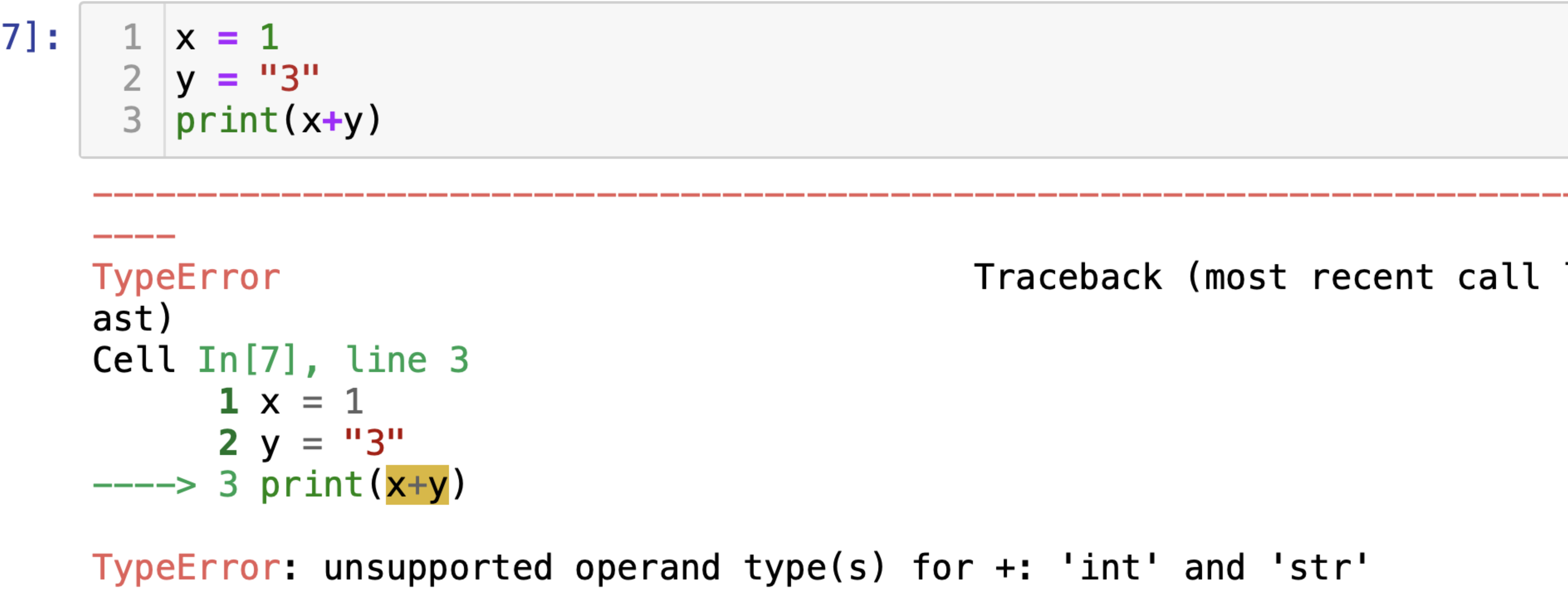
Sommerville, I. (2018) Engineering Software Products, An Introduction to Modern Software Engineering



Debugging

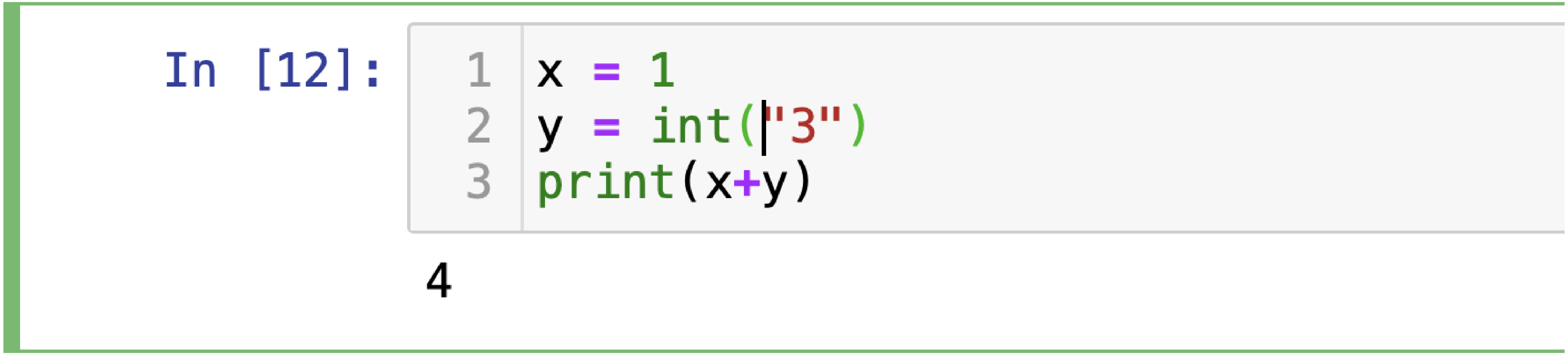


```
irror_mod = modifier_ob.
 mirror object to mirror_
mirror_mod.mirror_object =
operation == "MIRROR_X":
irror_mod.use_x = True
irror_mod.use_y = False
irror_mod.use_z = False
 _operation == "MIRROR_Y":
irror_mod.use_x = False
irror_mod.use_y = True
irror_mod.use_z = False
 _operation == "MIRROR_Z":
irror_mod.use_x = False
irror_mod.use_y = False
irror_mod.use_z = True

selection at the end -add
 _ob.select= 1
ier_ob.select=1
ntext.scene.objects.activ
("Selected" + str(modifier
irror_ob.select = 0
= bpy.context.selected_obj
ata.objects[one.name].sel

int("please select exactly

 OPERATOR CLASSES ----

types.Operator):
 X mirror to the selected
ject.mirror_mirror_x"
rror X"

context):
xt.active_object is not
```



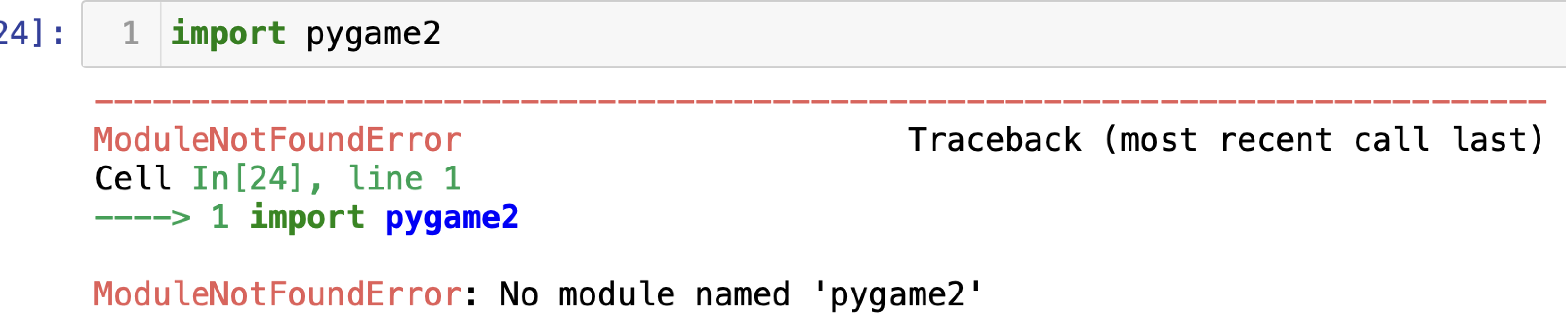
```
7]:
x = 1
y = "3"
print(x+y)
```

```
------------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call 
ast)
Cell In[7], line 3
      1 x = 1
      2 y = "3"
----> 3 print(x+y)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```



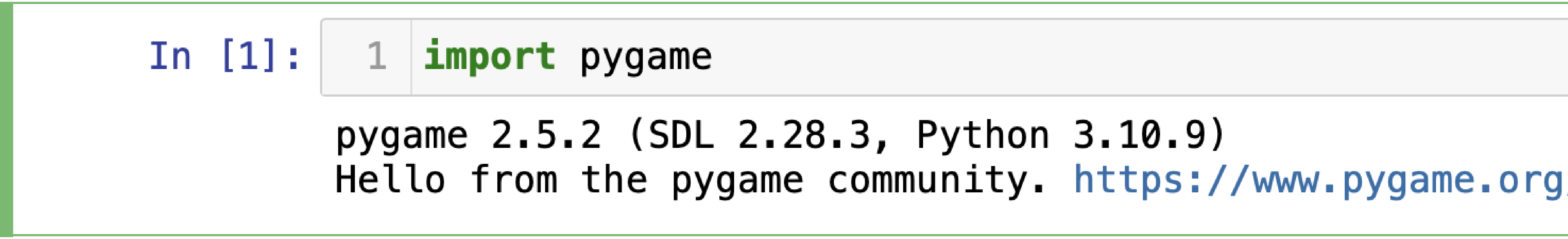
```
x = 1
y = int("3")
print(x+y)
```

```
4
```



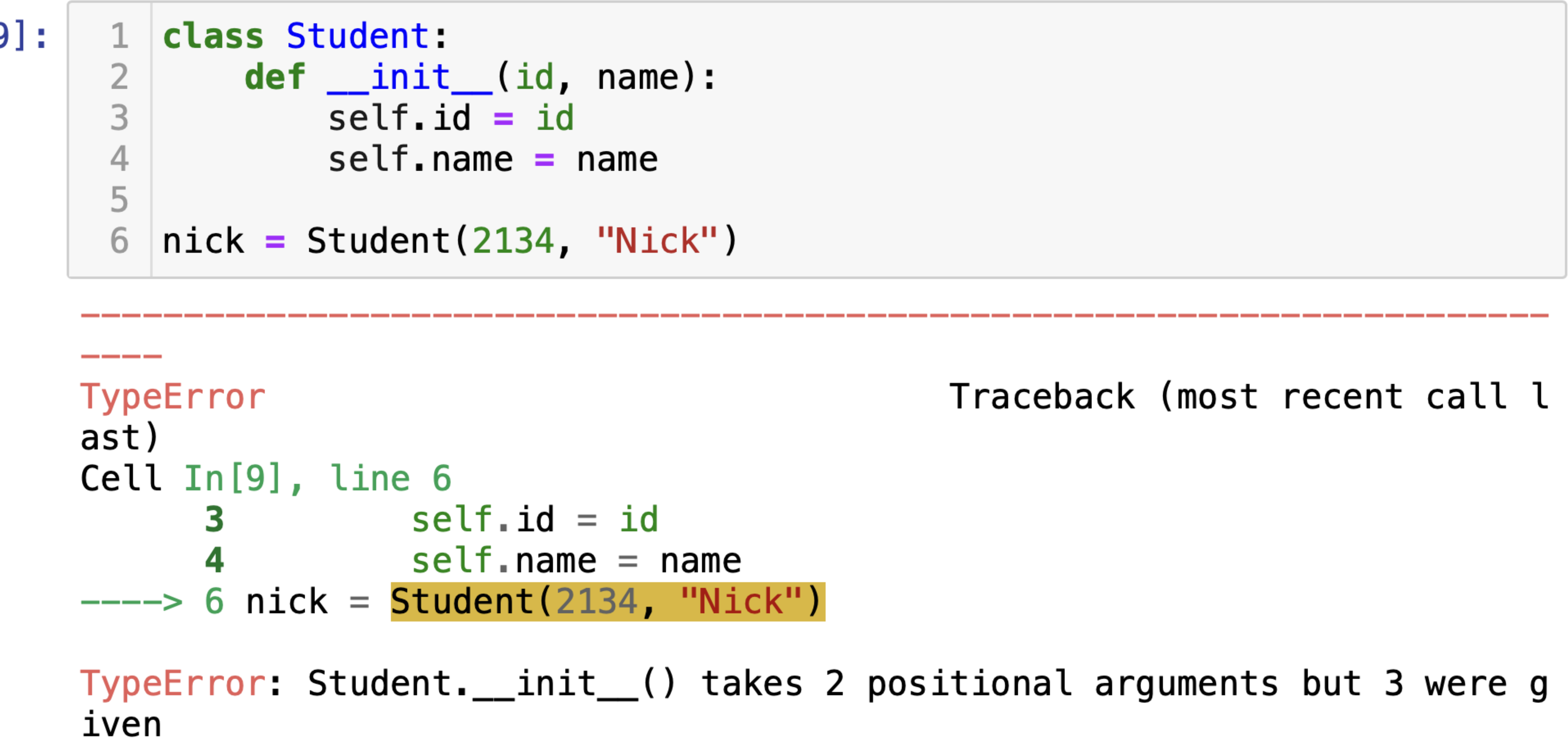
```
24]:  import pygame2
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[24], line 1
----> 1 import pygame2

ModuleNotFoundError: No module named 'pygame2'
```



```python
import pygame
```

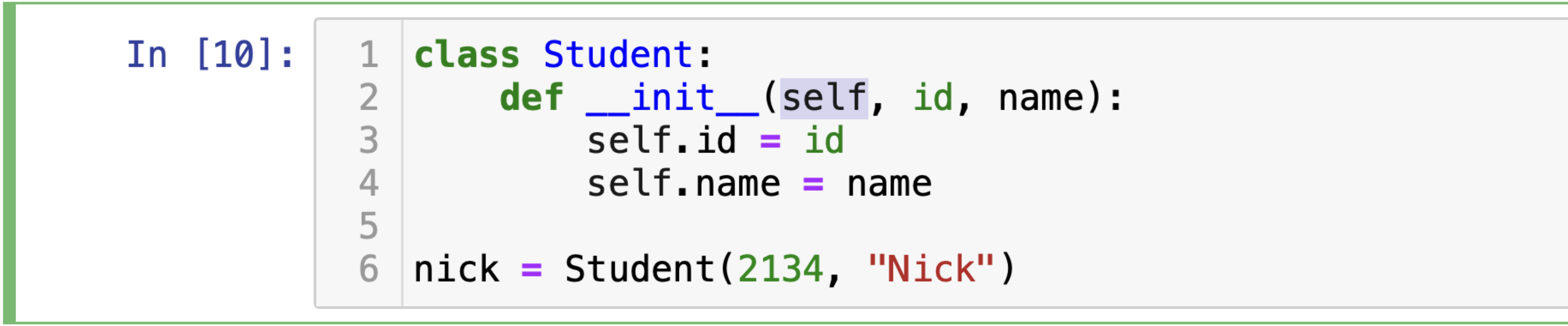
```
pygame 2.5.2 (SDL 2.28.3, Python 3.10.9)
Hello from the pygame community. https://www.pygame.org
```



9]:

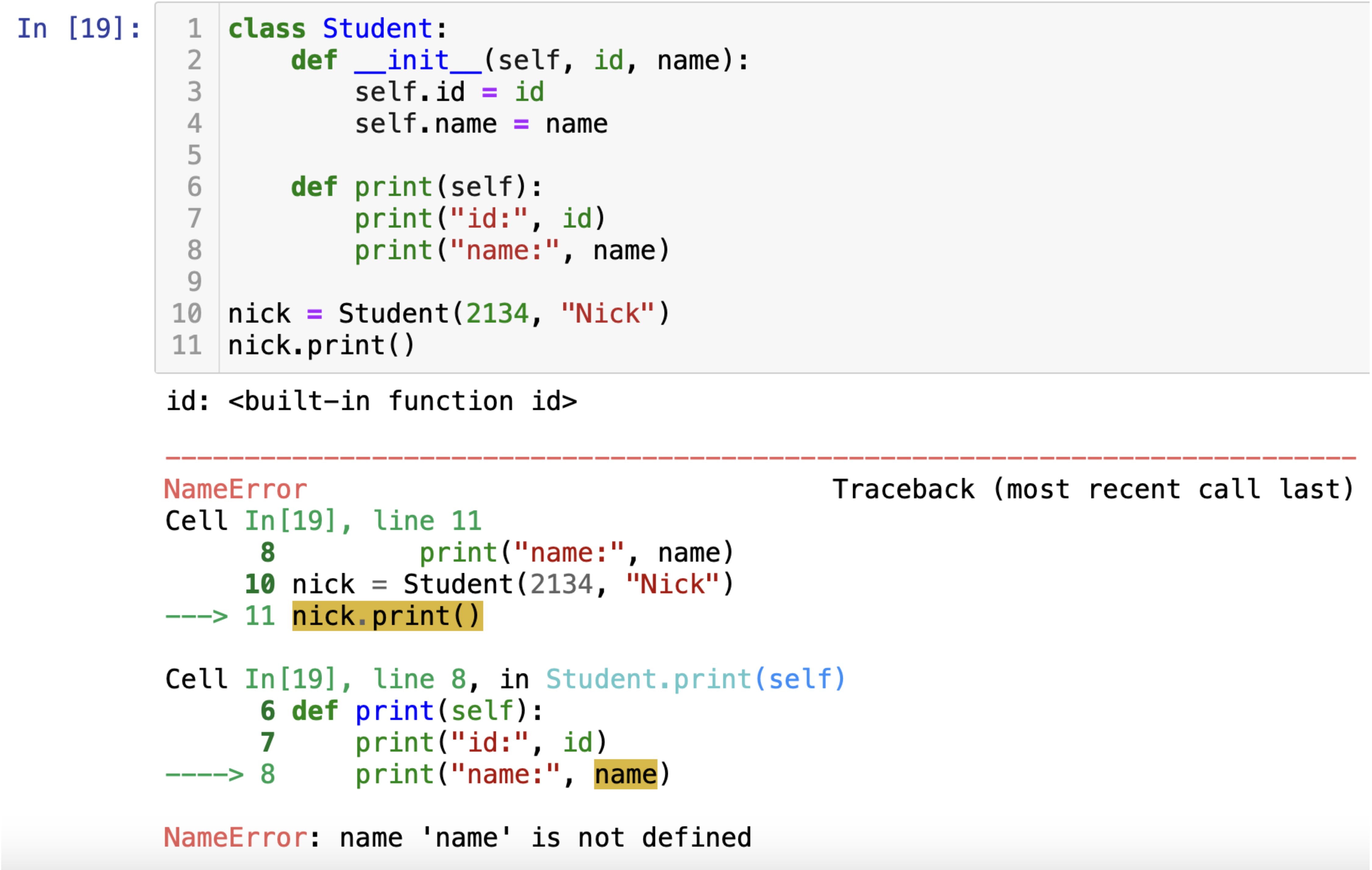
```
class Student:
    def __init__(id, name):
        self.id = id
        self.name = name

nick = Student(2134, "Nick")
```

```
---------------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
Cell In[9], line 6
      3         self.id = id
      4         self.name = name
----> 6 nick = Student(2134, "Nick")

TypeError: Student.__init__() takes 2 positional arguments but 3 were g
iven
```



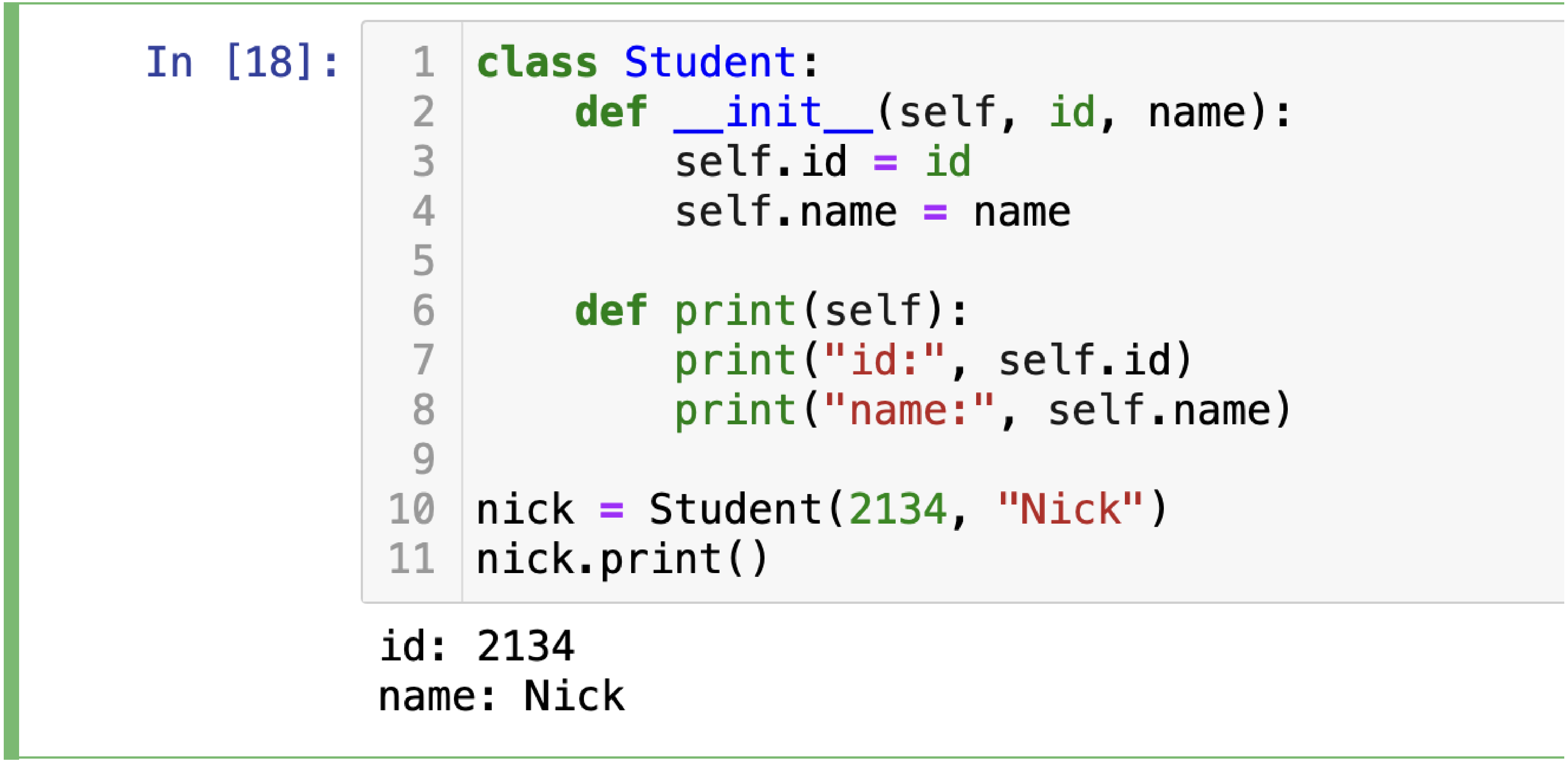
In [10]:

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

nick = Student(2134, "Nick")
```



```
In [19]:  1 class Student:
          2     def __init__(self, id, name):
          3         self.id = id
          4         self.name = name
          5 
          6     def print(self):
          7         print("id:", id)
          8         print("name:", name)
          9 
         10 nick = Student(2134, "Nick")
         11 nick.print()
```

```
id: <built-in function id>

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[19], line 11
      8         print("name:", name)
     10 nick = Student(2134, "Nick")
---> 11 nick.print()

Cell In[19], line 8, in Student.print(self)
      6 def print(self):
      7     print("id:", id)
----> 8     print("name:", name)

NameError: name 'name' is not defined
```



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

    def print(self):
        print("id:", self.id)
        print("name:", self.name)

nick = Student(2134, "Nick")
nick.print()
```

```
id: 2134
name: Nick
```