

클린코드-1부

- **클린코드의 정의와 학습내용**

- 가독성이 높음 --> Explicit Code와 추상화 (Day 1)

- 오류발생 가능성이 적음 --> 방어적 프로그래밍 (Day 1)

- 유지보수가 쉬움 --> 좋은 OOP 설계 (Day 2)



가독성을 위한 Naming

적절한 naming 은 의도를 잘 표현해준다

의도를 파악하기 어려운 코드

코드가 간단해서 동작은 파악이 되지만 의도를 알 수 없는 코드가 있다.
중요한 것은 **의도를 코드에 잘 표현**하는 것이다.

```
def run(the_list: list):  
    for t in range(len(the_list)):  
        if the_list[t] == 0:  
            continue  
        the_list[t] = the_list[t] - 1
```

run 함수를 호출하면, the_list 의 각 요소들의 값이 1씩 감소한다
동작에 대한 정보 외에 다른 정보를 알 수 없다

스토리 : 호텔 남은 기간

배열 각 칸은, 호텔 방의 남은 사용 일수를 뜻한다.

x는 0이다.

하루가 지날 때 마다 1씩 깎이고,
x는 사용기간이 끝난 일을 나타낸다.

| | | | | | | |
|---|---|---|---|---|---|---|
| x | x | 4 | 2 | x | x | 1 |
|---|---|---|---|---|---|---|

하루가
지나면

| | | | | | | |
|---|---|---|---|---|---|---|
| x | x | 3 | 1 | x | x | x |
|---|---|---|---|---|---|---|

Naming이 지켜진 것일까?

적절한 Naming을 지어보자. 의도를 분명하게 나타낸다.

아래와 같이 구현 후, Refactor 기능을 사용하여 이름을 변경한다.

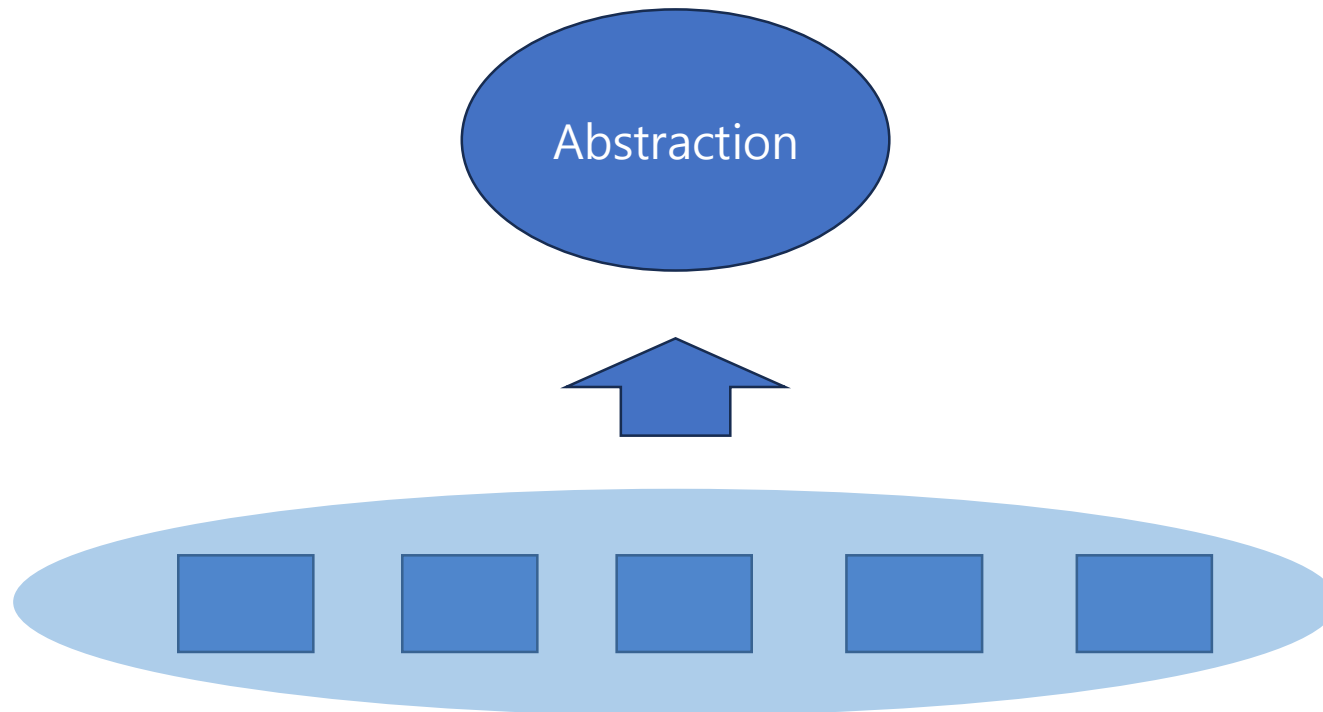
<https://github.com/jeonghwan-seo/Python-CRA-Example/blob/main/cleancode/%ED%98%B8%ED%85%94naming.py>

```
def run(the_list: list):  
    for t in range(len(the_list)):  
        if the_list[t] == 0:  
            continue  
        the_list[t] = the_list[t] - 1
```

추상화

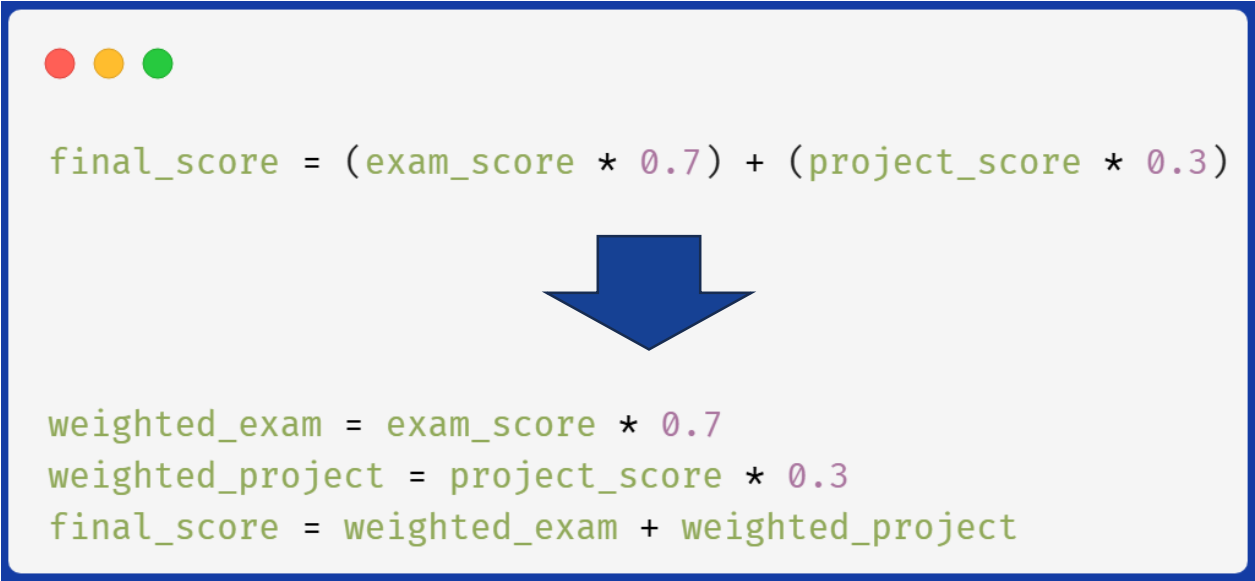
추상화는 불필요한 세부 정보들을 제거하고, 중요한 내용만을 드러내는 방식이다.

추상화를 통해 세부적인 내용 대신
의도가 잘 드러나는 코드를 구성할 수 있다.



표현식(expression) VS 변수(variable)

표현식이 복잡한 경우 가독성이 떨어진다. 변수를 도입해서 복잡한 표현식을 하나의 변수로 다루게 된다면 가독성이 좋도록 코드를 구성할 수 있다



```
final_score = (exam_score * 0.7) + (project_score * 0.3)
```

```
weighted_exam = exam_score * 0.7  
weighted_project = project_score * 0.3  
final_score = weighted_exam + weighted_project
```


함수 이름을 이용해서 표현

함수를 이용하면 동작의 세부사항을 감추고 동작을 요약하는 중요한 정보를 드러낼 수 있다. 이를 통해 호출 측은 가독성이 좋아진다.

```
scores = [95, 82, 67, 58, 74]
grades = []

for score in scores:
    if score >= 90:
        grade = "A"
    elif score >= 80:
        grade = "B"
    elif score >= 70:
        grade = "C"
    elif score >= 60:
        grade = "D"
    else:
        grade = "F"
    grades.append(grade)

print(grades)
```



```
def convert_all_scores_to_grades(scores):
    return [convert_score_to_grade(score) for score in scores]

# 호출 측
scores = [95, 82, 67, 58, 74]
grades = convert_all_scores_to_grades(scores)
print(grades)
```

Bad Naming - 1

흔한 약어

읽는 사람마다 해석이 다를 수 있다.

e.g.) ax, aix, sco, hp 등

시각적 혼란

e.g.) 대문자 I와 소문자 l, 대문자 O, 소문자 o

IIIIIIIIII 000Oo o O0 o oO0

의미 없거나 해석이 어려운 이름

e.g.) name_str : 여기서 str 은 불필요한 잡음(noise) / a1, a2, b1 등

발음하기 어려운 이름을 피하자.

e.g.) genymdhms

코드 작성시 단어 선택에 있어서 읽는 사람에게 혼란을 주어서 안된다.

Info / Data / Value 와 같은 단어는 의미 있는 구별이 안된다

- Product, ProductInfo, ProductData 가 만들어져 있다면 어떤 것을 사용할지 불명확
- Info, Data, Value 가 의미 있는 정보를 포함하지 않는다

Manager / Processor 와 같은 단어는 과도하게 많은 기능을 가진 클래스 암시

- 클래스는 너무 많은 기능을 가지고 있으면 안된다. (추후 SRP 에서 설명)
- 해당 이름은 너무 범용적인, 많은 기능을 가진 클래스를 암시한다.

naming 은 정보 전달이다.

1. 정확하게 전달한다

naming 을 통해 오해가 없어야 한다

e.g. temp -> 의미 모호, email_notification_enabled -> 정확하게 의미 전달

2. 의도를 명시한다

변수나 함수명을 통해 코드 그 자체로 의도를 파악할 수 있도록 해야 한다

e.g. get_data() -> fetch_users_from_db() 가 의도를 훨씬 잘 전달한다

3. 일관성 있는 규칙을 지켜 파악하기 쉽게 해야 한다.

일관성이 있는 naming 은 비슷한 개념들을 파악하기 쉽게 해준다

같은 개념에는 동일한 이름을 반복적으로 사용한다

e.g. fetch, get, retrieve -> 일관성이 없다. 하나로 통일하는 게 좋다.

알고리즘, 패턴 이름을 사용해서 명확해 진다면 이를 알 수 있는 이름을 짓는다
프로그래머들은 Computer Science 용어를 이미 숙지하고 있는 경우가 많다

Controller, Manager, Driver 등 용어를 사용할 때 통일이 필요하다

AccountManager, AccountController, AccountProcessor 이름을 보고 이 셋을 구별할 수 없다
따라서 팀 내에서 통일이 되어야 한다.

검색하기 좋도록 이름을 짓는다

명확하고 고유한 이름은 IDE 에서 검색, 추적, 리팩토링이 가능하게 해준다

꼭 필요한 정보를 함축적으로 잘 담아 전달해야 하며,
의미 있도록 구분이 잘 되도록 naming 해야 한다.

PEP 8 naming 권장 규칙

PEP 8 을 지키면 일관성 있는 naming 을 할 수 있다.

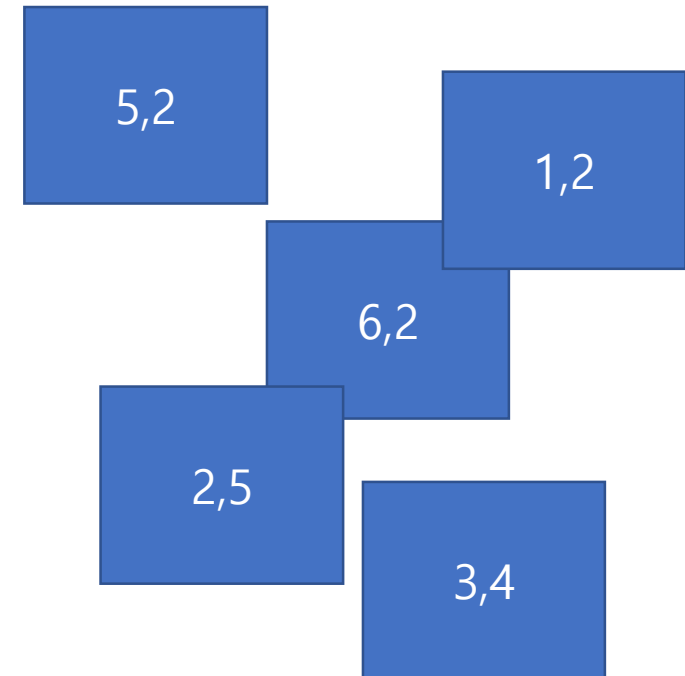
1. 함수, 변수, 속성, 모듈명 : lowercase_with_underscores 형식
calculate_total_price, total_price, self.order_amount, data_loader.py
2. 클래스명 및 예외명: CapitalizedWords (예외명 + Error)
HttpRequest, DataValidationError
3. 상수 : ALL_CAPS_WITH_UNDERSCORES
MAX_RETRIES, DEFAULT_PORT
4. self/cls : 인스턴스 메서드 첫 인자 / 클래스 메서드 첫 인자 (필수)

함수는 동사 + 목적어 / 클래스는 명사

함수는 본질적으로 동작이므로 동사 + 목적어로 나타내고,
클래스는 명사를 사용한다.

스토리 : Draw Button Machine

점을 찍을 좌표를 배열에 저장해두고,
Draw 버튼을 누르면,
점 위치에 버튼을 찍어주는 프로그램



[도전] Draw Machine Naming 변경

이름을 수정해보자.

<https://github.com/jeonghwan-seo/Python-CRA-Example/blob/main/cleancode/drawmachine.py>

함수명은 동사가 좋다.

함수는 본질적으로 동작이기 때문

이름을 변경할 때는 단축키를 이용한다

```
class xy_info:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class xyManager:
    def __init__(self):
        self._arr = []

    def add(self, data: xy_info):
        self._arr.append(data)

    def drawing(self):
        for o in self._arr:
            print(f"{o.x}, {o.y} 에 버튼 찍힘")

a = xyManager()
a.add(xy_info(1, 2))
a.add(xy_info(2, 3))
a.add(xy_info(5, 5))
a.add(xy_info(7, 1))
a.drawing()
```



가독성을 위한 Method 작성

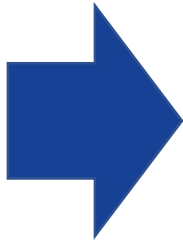
함수는 추상화의 첫 단계

긴 함수는 이해하기 어렵다

긴 함수는 이해하기 어렵다. 아래 코드를 읽고 잠시 해석해보자

<https://gist.github.com/mincoding-jh/3df1d8592b0ade1bc19e025cdb60314c>

```
def testableHtml(pageData, includeSuiteSetup):
    wikiPage = pageData.getWikiPage()
    buffer = []
    if pageData.hasAttribute("Test"):
        if includeSuiteSetup:
            suiteSetup = PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_SETUP_NAME, wikiPage)
            if suiteSetup != None:
                pagePath = suiteSetup.getPageCrawler().getFullPath(suiteSetup)
                pagePathName = PathParser.render(pagePath)
                buffer.append("!include -setup .")
                buffer.append(pagePathName)
                buffer.append("\n")
            setup = PageCrawlerImpl.getInheritedPage("Setup", wikiPage)
            if setup != None:
                setupPath = wikiPage.getPageCrawler().getFullPath(setup)
                setupPathName = PathParser.render(setupPath)
                buffer.append("!include -setup .")
                buffer.append(setupPathName)
                buffer.append("\n")
            buffer.append(pageData.getContent())
        if pageData.hasAttribute("Test"):
            teardown = PageCrawlerImpl.getInheritedPage("TearDown", wikiPage)
            if teardown != None:
                teardownPath = suiteSetup.getPageCrawler().getFullPath(teardown)
                teardownPathName = PathParser.render(teardownPath)
                buffer.append("!include -teardown .")
                buffer.append(pagePathName)
                buffer.append("\n")
        if includeSuiteSetup:
            suiteTeardown = PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_TEARDOWN_NAME, wikiPage)
            if suiteTeardown != None:
                pagePath = suiteSetup.getPageCrawler().getFullPath(suiteTeardown)
                pagePathName = PathParser.render(pagePath)
                buffer.append("!include -teardown .")
                buffer.append(pagePathName)
                buffer.append("\n")
        pageData.setContent("".join(buffer))
    return pageData.getHtml()
```



```
def render_page_with_setups_and_teardowns(page_data, is_suite):
    is_test_page = page_data.has_attribute("Test")
    if is_test_page:
        test_page = page_data.get_wikiPage()
        new_page_content = ""
        new_page_content = include_setup_pages(test_page, new_page_content, is_suite)
        new_page_content += page_data.get_content()
        new_page_content = include_teardown_pages(test_page, new_page_content, is_suite)
        page_data.set_content(new_page_content)
    return page_data.getHtml()
```

```
def render_page_with_setups_and_teardowns(page_data, is_suite):
    if is_test_page(page_data):
        include_setuppages_and_teardownpages(page_data, is_suite)
    return page_data.get_html()
```

복잡

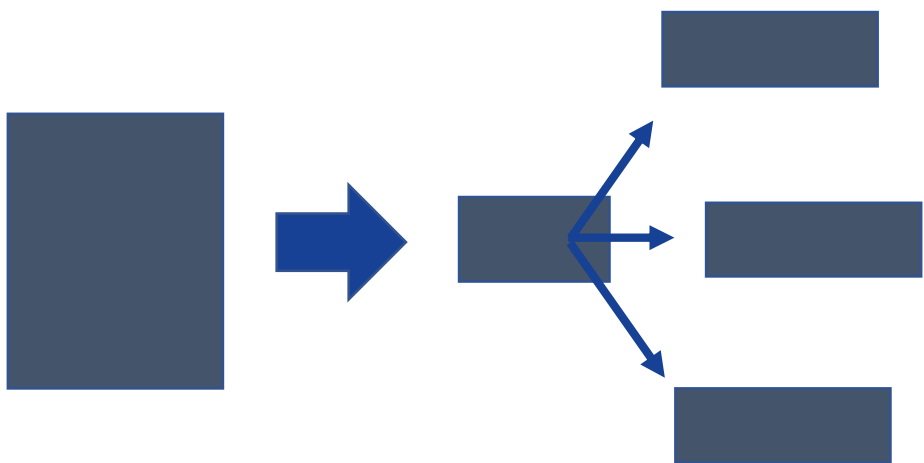
심플

함수를 작게 만든다

객관적인 근거를 대기는 어렵지만, 함수의 크기와 복잡도는 상관성이 있다.

따라서 함수의 크기가 작을수록 이해하기 좋다.

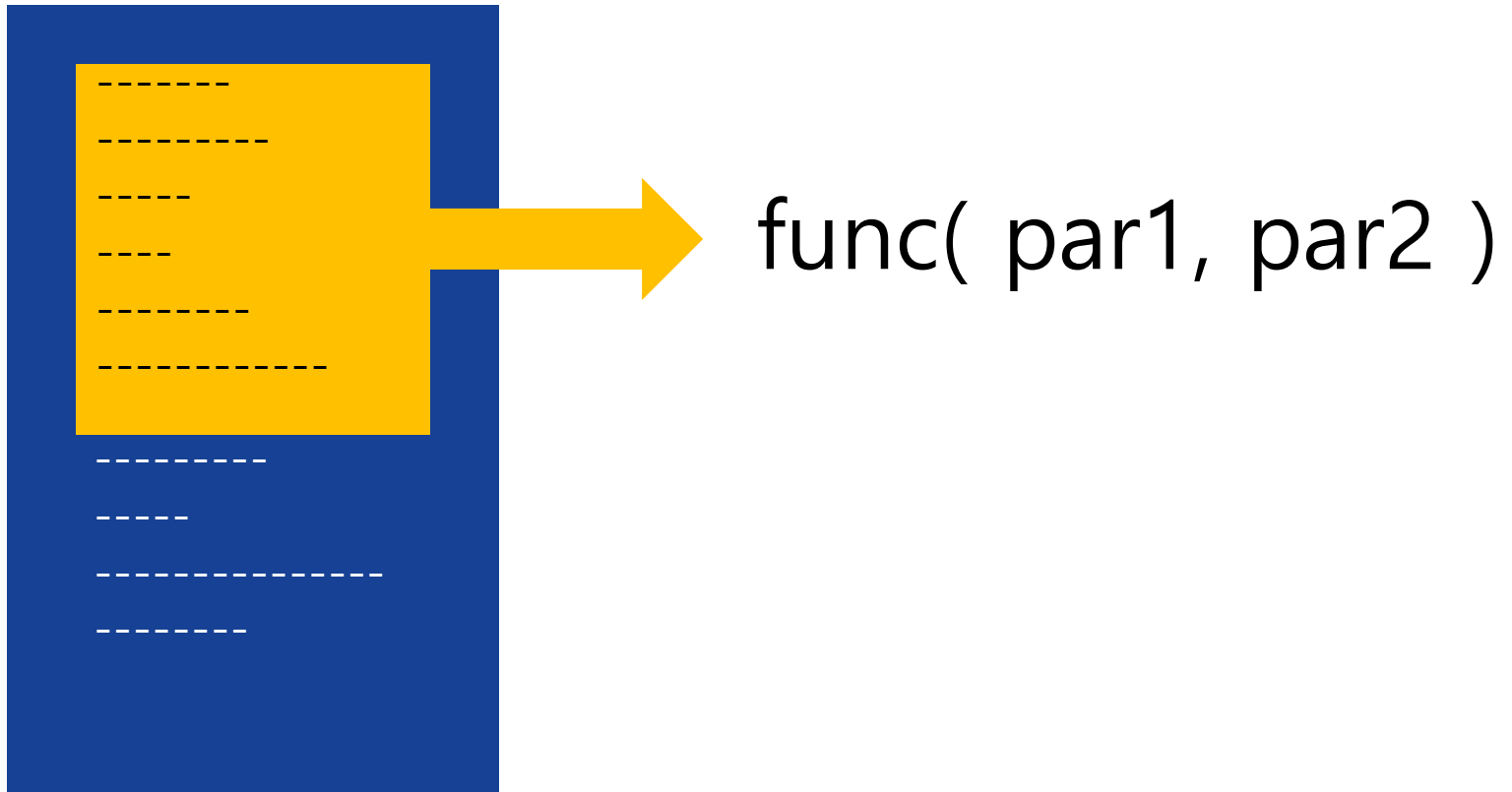
많은 오픈소스 프로젝트가 함수 하나에 평균 8~13줄 정도의 코드를 가진다는 분석이 있음



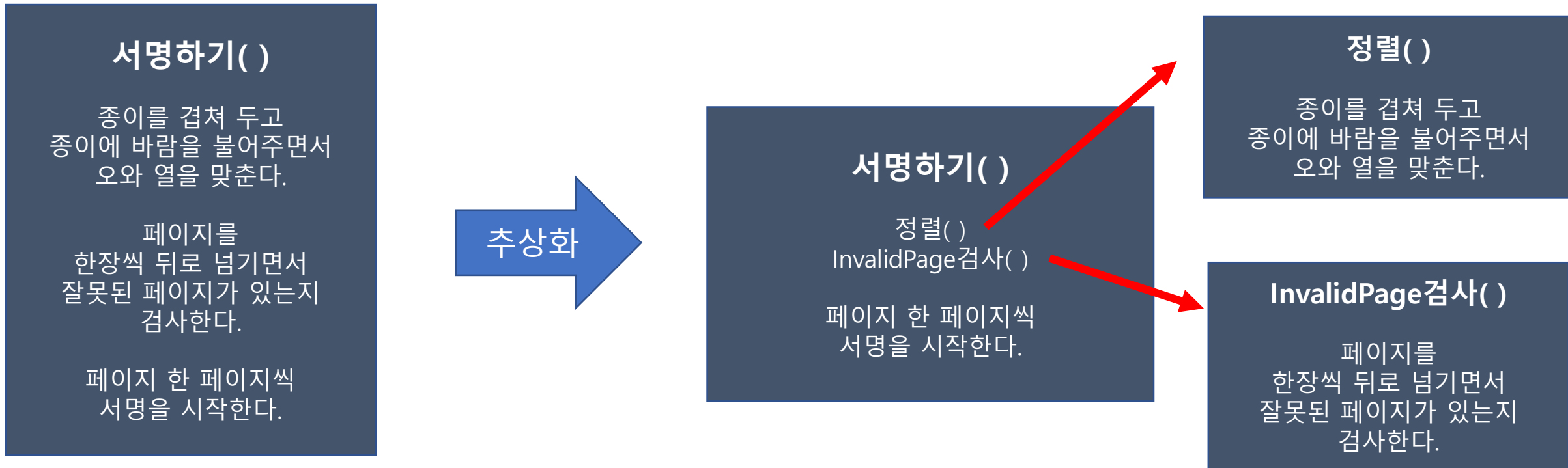
| Package | License | Line count | Docstrings (% of lines) | Comments (% of lines) | Blank lines (% of lines) | Average function length |
|----------|--------------|------------|----------------------------|--------------------------|-----------------------------|----------------------------|
| HowDol | MIT | 262 | 0% | 6% | 20% | 13 lines of code |
| Diamond | MIT | 6,021 | 21% | 9% | 16% | 11 lines of code |
| Tablib | MIT | 1,802 | 19% | 4% | 27% | 8 lines of code |
| Requests | Apache 2.0 | 4,072 | 23% | 8% | 19% | 10 lines of code |
| Flask | BSD 3-clause | 10,163 | 7% | 12% | 11% | 13 lines of code |
| Werkzeug | BSD 3-clause | 25,822 | 25% | 3% | 13% | 9 lines of code |

복잡한 내용을 의미있는 덩어리(chunk)의 함수로 추출하여 분리할 수 있다

추상화, 추상화의 결과물 = Abstraction



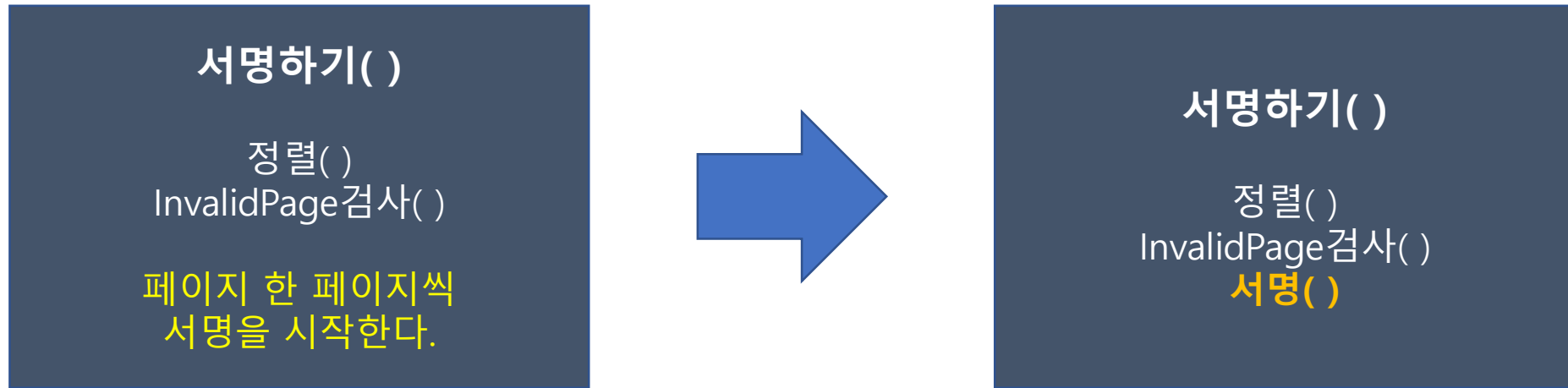
코드를 이해하는데 중요하지 않는 부분은 함수로 빼서 추상화하자



추상화 Level 맞추기

일부만 추상화를 하는 것이 아니다.

추상화를 한다면 주변 코드가 개념적으로 같은 수준(Level)이 되도록 한다.



코드의 논리적 흐름이 안 끊기고 쉽게 따라갈 수 있게 해준다.
각 기능별로 어떤 수준에서 동작하는지가 명확해진다.

거대하지 않는, 가독성 있는 makeSign() 메서드 만들기

https://github.com/jeonghwan-seo/Python-CRA-Example/blob/main/cleancode/dosign_methodextract1.py

```
class Node:
    def __init__(self, date_code, name):
        self.date_code = date_code
        self.name = name
        self.is_signed = False

    def do_sign(self):
        self.is_signed = True

    def print_sign(self):
        print(f"{self.date_code} : {self.name}")

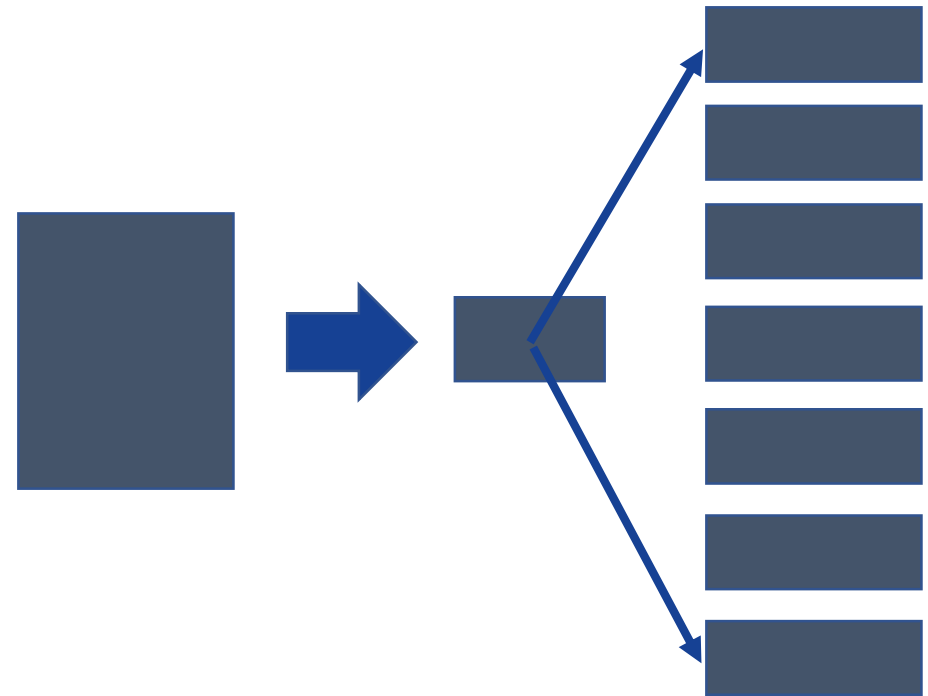
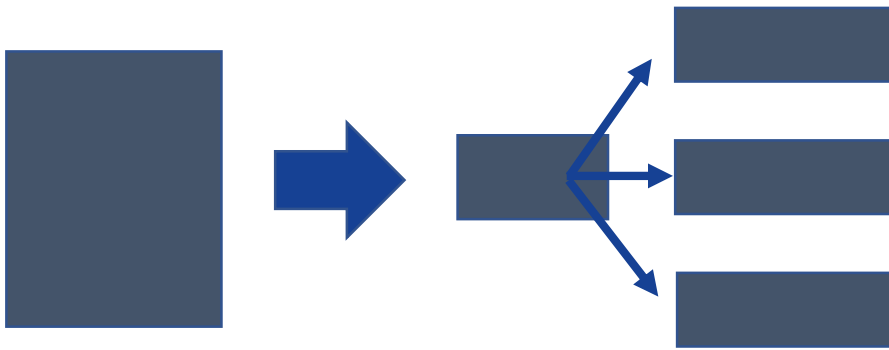
class Sign:
    def make_sign(self, sign_lst: List[Node]):
        # 1. 서명 정렬하기
        for y in range(len(sign_lst)):
            for x in range(y + 1, len(sign_lst)):
                if sign_lst[y].date_code > sign_lst[x].date_code:
                    sign_lst[y], sign_lst[x] = sign_lst[x], sign_lst[y]

        # 2. valid 검사
        flag = False
        for tar in sign_lst:
            if 0 < tar.date_code < 10: continue
            flag = True
            break

        if flag is True:
            raise Exception()
        else:
            # 3. 서명 하기
            for tar in sign_lst:
                tar.do_sign()
                tar.print_sign()
```


Sub 함수로 추출 시 주의사항

함수를 너무 작게 나누게 된다면 각 함수들은 너무 의미가 없는 덩어리(chunk)가 되고, 분리된 함수로 인해 흐름 따라가기 어려워진다.



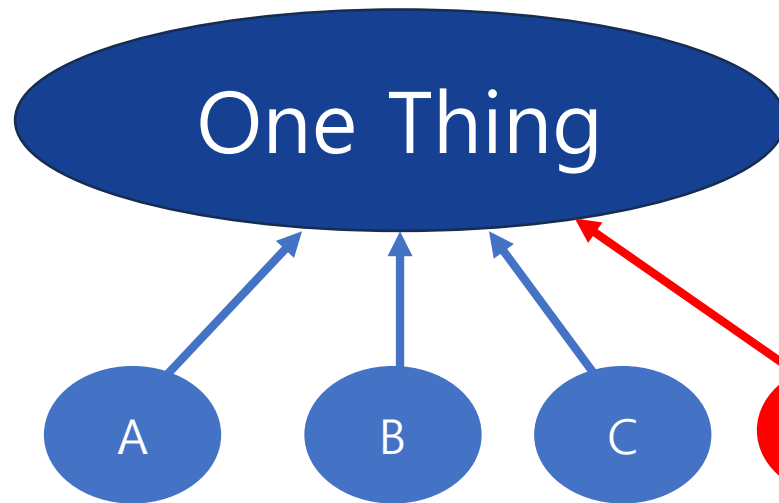
오히려 복잡해지고 유지보수하기 어려워진다.

Do One Thing! 한 가지를 잘해야 한다

함수는 작고, 하나의 일만 해야 한다. – Robert C. Martin

“함수는 한 가지만 해야한다.”

- CPU가 한 가지 Action만을 수행하도록 함수를 작성하라는 의미가 아니다.
- 의미적으로 하나의 책임을 수행한다는 것
- 함수 내부 코드가 하나의 목적을 향해 있으면 “한 가지 일”로 볼 수 있다.



One Thing 에 부합하지 않는 내용이 있다면, 분리해야한다!

다음과 같이 함수를 추출하는 경우들을 분류할 수 있다.

1. Subtask 로써의 함수 추출을 할 수 있다.

부모함수가 추출된 함수를 호출한다. 주로 helper method(function) 라 불리며,
이 같은 함수들은 원래 함수의 **각 부분 기능들을 명확히** 해주고 **가독성을 높여준다**.

2. 한 함수에서 여러가지 작업을 하는 함수라면 분리된 함수로 쪼갬다.

하나의 함수가 서로 다른 일을 한다고 판단되면, **명확한 경계로** 분리한다.

3. 함수를 통해 중복된 코드를 관리, 재사용할 수 있다.

중복된 코드를 함수로 만들어두면 관리, 시스템 이해에 도움이 된다.
또한 비슷한 코드의 재사용성을 갖추게 된다.

함수 이름 잘 짓기

서술적인 이름을 사용하자

- 이름이 길어도 된다. (팀마다 다를 순 있다)

의도를 잘 나타내는 긴 이름은 다음 항목보다는 더 좋다

- 짧고 의미를 알 수 없는 이름
- 함수를 설명하는 긴 주석

예시

- test → isTestable
- insertPart → includesSetupAndTeardownPages

함수 인수는 최소한으로 줄이는 것이 코드를 이해하기 좋다.

- `include_setup_page_info(content)` **vs** `include_setup_page()`
- 인수 0개가 가장 이상적, 인수 1개도 이해하기 쉽다 ex) `render(pageData)`

함수 시그니처(Signature)는 명확하고 단순해야 한다.

- 너무 많은 인자나, 순서가 헷갈리는 인자들을 사용할 경우 사용자가 혼란을 겪는다
- 적절히 구조화하거나 객체로 묶는 것이 좋다

함수 인수를 더 줄이도록 노력하자

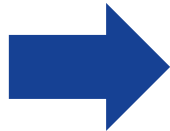
함수 인수 2개부터 이해하는데 시간이 더 걸린다.

`write_field(name)` vs `write_field(output_stream, name)`

인수 객체를 쓰자(파라미터 객체 도입)

- `make_circle(x, y, radius)`
→ `make_circle(point, radius)`

```
def make_circle(x, y, radius):  
    pass
```

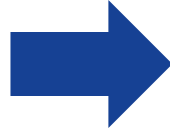


```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
def make_circle(point: Point, radius):  
    pass
```

함수 인수에 flag

함수 안에서 두 가지 동작을 구현해야 하는 flag 인자는 피하자.

```
def render(flag):  
    if flag:  
        arrange_text()  
        visual_dom_object()  
    else:  
        unarrange_text()  
        porting_object()
```



```
def enable_render():  
    arrange_text()  
    visual_dom_object()  
  
def disable_render():  
    unarrange_text()  
    porting_object()
```

결론. 소프트웨어를 작성하는 것은 글짓기와 같다.

글짓기 할 때는, 초안을 작성하고 읽기 좋게 다듬는다.

- 초안은 대개 서투르고, 어수선하다.
- 읽는 사람이 내 의도를 정확히 파악할 지 생각해보며, 반복적으로 읽고 글을 고친다.

코드 작성 후, 글짓기처럼 다듬는 작업이 필요하다.

- 코드를 다듬고, 추상화하고, 이름을 바꾸며, 중복을 제거한다
- 메서드 순서도 바꾸고 클래스를 쪼개거나 합치곤 한다
- 테스트는 항상 통과하면서 지속적으로 코드를 고친다

처음부터 Clean Code가 작성되길 기대하지 말자.
하지만 최종적으로 수정하다 보면 Clean Code가 완성된다.



Explicit 코드

"Explicit is better than implicit." – PEP20(The Zen of Python)

코드 읽는 사람이 **직관적**으로 이해할 수 있는 코드, **의도가 분명하게** 드러나는 코드를 뜻한다.

PEP20 (The Zen of Python) 에는 다음과 같은 구절이 등장한다.

"Explicit is better than implicit."

누가 봐도 **명확하게 의도를 잘 드러내야**

1. 의도와 다르도록 추측하지 않기에 가독성이 좋아진다
2. 숨기는 바가 없으므로 오류 발생시 추적이 쉽다
3. 작성 이유가 명시적이므로 코드 리뷰 시 불필요한 질문이 줄어든다

하나의 명시적 코드 작성

문제를 해결할 **명백한 방법은 하나**이면 좋다.

같은 의미를 갖지만 가독성이나 명시성이 분명한 것으로 작성해야 한다

- **if not a is b VS if a is not b**

긍정적인 식을 부정한 것 (not (a is b))

부정을 내부에 넣은 식 (a 는 b 가 아니다 로 바로 해석)

- **빈 컨테이너, 시퀀스 ([], " 등)을 검사**

if len(lst) == 0 VS if not lst

if len(lst) > 0 VS if lst

- **x == None VS x is None**

== 은 동등(equality)한가 검사하는 것, **is** 는 객체의 정체성(identity)을 비교

- **dict[key] if key in dict else default VS dict.get(key, default)**

dictionary 의 기능을 이용하는 것이 좀 더 간결하고 명시적

타입 힌트

python 은 동적타입이지만, type hint(type annotation)를 통해 명시적인 코드로 만들 수 있다.

```
def greet(name: str) -> str:  
    return f'Hello, {name}'  
  
greet("hwan")  
greet(123)
```

type hint 를 이용하면
정적타입검사를 할 수 있다

```
Money = float  
def sum_price(value: Money) -> Money:  
    return value + 10.0
```

type alias 를 이용해서 표현력을 높일 수 있다

함수를 사용하는 입장에서, 숨겨진 내부 구현을 모르고도 정확히 사용할 수 있어야 한다.

내부 로직은 숨기되, 무엇을 하는지를 잘 드러내야 한다.

명시하지 않은 부분으로 호출자들의 실수가 발생해서는 안된다.

- `def make_dict(x,y)` vs `def make_dict(*args)`

`*args` 는 유연하지만 개수, 파라미터명 등이 모호하다. 개별인자가 오히려 더 명확하다

- 키워드 인자를 사용해서 명시적인 사용 유도


`def move(x,y,*speed): ...` 을 호출하는 경우 `move(1,2,speed = 10)` 이런식으로

`*` 뒤로는 키워드로 명시해서 사용해야 한다

Uniform Access Principle

메서드를 속성처럼 보이게 할 수 있다.

즉, 속성과 메서드는 동일한 방식으로 접근하도록 할 수 있다. (Uniform Access)
이렇게 하면 사용자는 필드(field)인지 계산된 결과인지 모르도록 할 수 있다.

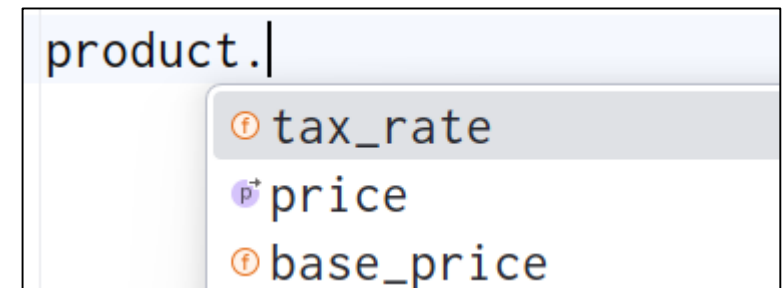


```
class Product:
    @property
    def price(self):
        return self.base_price * (1 + self.tax_rate)

product = Product()
print(product.price)
```

메서드지만 속성처럼 사용 가능

속성처럼 보이지만 내부는 계산 작업



```
product.|
```

- tax_rate
- price
- base_price

IDE에서 field 인지 property 인지 구분해준다

속성인가? 계산인가?

복잡하거나 비용이 큰 계산 혹은 side effect가 있는 계산이라면 속성처럼 보이는 것보다, 오히려 명시적인 naming 이 좋다.



```
class Product:
    @property
    def price(self):
        # 계산식이 단순하다

    def compute_discounted_price(self):
        # 비용이 큰 계산 or side-effect 가 있는 계산
```

명시적인 naming 으로 드러낸다

감사합니다.