

좋은 코드를 작성하려면?

...

한기용

Table of Contents

1. 효과적인 학습이란?
2. 기본 닦기
3. 좋은 코드란?
4. 코딩 규칙과 모범 사례 따르기
5. 코드 리뷰하기
6. 코드 테스트 (Unit Test)
7. 코드 테스트 (Unit Test) 실습
8. 소스 버전 관리 툴 익히기
9. CI/CD 프로세스 만들기
10. 계속 학습하기
11. 추천 자료

효과적인 학습이란?

새로운 주제의 효과적인 학습방법 - 긍정적인 감정 이입하기

- 잘 모르는 용어나 개념도 반복해서 듣고 사용하다 보면 익숙해짐
- 스트레스 받지 말고 앞으로 친해지겠다는 생각하기

안녕, 반가워!
처음이라 아직 내가 잘 몰라.
다음에는 더 친해지자!
잘 부탁해

긴 호흡으로 바라보기

- 지금 공부해야 할 내용이 굉장히 많다보니 시간이 부족할 수 있음
- 한번 뒤흔쳤다고 포기하지 말고 늦게라도 계속해서 따라가기
- 경우에 따라서는 2번 듣는다고 생각하기
 - 처음 한번 강의 내용을 빠르게 듣고 다시 한번 조금 느리게 따라가보기

그냥 열심히 하기

- 내가 잘 하고 있는지 너무 생각이 많으면 힘들어짐
- 스톡데일 패러독스
 - 큰 방향을 가지고 하루 하루 열심히 하기!



기본 닦기

좋은 개발자의 시작은? (행동적인 부분)

- 긍정적인 자세
- 남과 비교하지 않기
- 커리어를 길게 보기
- 질문 잘하기
- 의사 소통 잘하기
- 문제 정의 잘하기
- 최소 비용으로 최대 결과 내기
- 결과 내기

좋은 개발자의 시작은? (기술적인 부분)

- 프로그래밍 언어 선택하기
- 구문 및 데이터 유형 이해하기
- 깔끔하고 읽기 쉬운 코드 작성
- 생산성 툴 익히기
- 문제해결 하기

프로그래밍 언어 선택하기

- 파이썬
- 스칼라
- 자바
- 코틀린
- 자바스크립트
- ...

구문 및 데이터 유형 이해하기

- 기본 문법 이해 -> 고급 문법 이해
- 상황에 맞게 적합한 구문과 데이터 유형 사용
- 많이 코딩해보기 (그러면서 고민해보기)
- 남의 코드 보고 배우고 내 코드 리뷰 받기

깔끔하고 읽기 쉬운 코드 작성

- 이번 톡의 핵심
- 이것 역시 많이 코딩해보면서 고민하기와 코드 리뷰 받기가 핵심
 - a. 팀에 시니어가 없다면 오픈소스 코드 보기
 - b. 챗GPT등에게 코드 리뷰해달라고 하기 (생산성 툴을 최대한 이용하기)

생산성 툴 익히기

- Git과 Github
 - a. 커맨드라인에서 자유롭게 커밋하고 브랜치 만들고 PR 만들기
- 소스코드 에디터 (IDE: Integrated Development Environment)
 - a. 단축키 많이 사용해보기
 - i. VS Code Windows 단축키
 - b. 플러그인 많이 사용해보기
 - i. 유용한 VS Code Plugin
 - ii. 여유가 되면 Github Copilot 같은 플러그인 설치하기

좋은 코드란?

좋은 코드를 작성하는 것은 훌륭한 개발자의 필수 요소

- 깔끔한 코드는 읽고, 이해하고, 수정하기 쉬움
- 코드는 명확한 이름, 일관된 형식, 의미 있는 주석으로 체계적으로 구성
- 좋은 코드는 테스트가 가능하며 코드와 함께 유닛 테스트를 작성해야 함
- 클래스와 함수는 분명한 하나의 일을 하도록 구현해야 함
- 코드는 모듈화되어야 하며 중복을 피해야 함
- 오류 처리는 코드베이스 전체에서 철저하고 일관성 있게 이루어져야 함 (로깅)
- 코드는 작성자가 아닌 다른 개발자들을 염두에 두고 설계해야 함
- 단 중요한 일에 더 완벽을 기할 것

코드 작성 원칙

- DRY (Don't repeat yourself)
- KISS (Keep it simple, stupid)

코드 작성 원칙 - DRY (Don't Repeat Yourself)

- 같은 코드를 여러 번 작성하고 있다면? 시간 낭비!
 - 한 번 작성하여 필요한 곳에 재사용할 수 있는 방법을 찾아야 한다!
-
1. 반복되는 작업이라면 함수로 만들 것
 2. 동일한 일이 반복된다면 루프로 만들 것
 3. 비슷한 클래스를 반복해서 만든다면 클래스 계승 구조를 사용해볼 것

코드 작성 원칙 - **DRY**가 깨진 경우

화씨에서 섭씨로 온도 변환

```
f_temp = 55
```

```
k_temp = (f_temp - 32) * (5 / 9)
```

```
f_temp2 = 46
```

```
k_temp2 = (f_temp2 - 32) * (5 / 9)
```

```
f_temp3 = 56
```

```
k_temp3 = (f_temp3 - 32) * (5 / 9)
```

3개 변수 출력

```
print(month1)
```

```
print(month2)
```

```
print(month3)
```

코드 작성 원칙 - DRY를 적용한 경우

```
def fahr_to_celsius(fahr)
    """Convert temperature in Fahrenheit to Celsius.

    Parameters:
    fahr (int or float): the temperature in Fahrenheit.

    Returns:
    celsius (int or float): the temperature in Celsius.
    """
    celsius = ((fahr - 32) * (5 / 9))
    return celsius
```

```
fahr_to_celsius(55)
fahr_to_celsius(46)
fahr_to_celsius(56)
```



```
# 3개 변수 출력
all_vars = [month1, month2, month3]
for avar in all_vars:
    print(avar)
```

```
all_fahrs = [55, 46, 56]
for f in all_fahrs:
    fahr_to_celsius(f)
```

코드 작성 원칙 - KISS (Keep It Simple Stupid)

- 불필요한 복잡성을 피하고 코드를 가능한 한 단순하게 유지
 - 코드를 꼭 짧게 작성해야한다는 것은 아님 (특히 람다 함수)
 - 별다른 이유가 없다면 내장 함수 사용
- 너무 많은 기능을 하나의 함수로 구현하지 말것
- 한 함수가 너무 길어지면 이를 다수의 함수로 나눌 것
- 결국 읽기 쉽고 이해하기 쉬운 코드를 작성하기 위해 노력
 - 다른 사람들이 이해하기 쉽고 테스트가 쉬워짐

코드 작성 원칙 - **KISS**가 깨진 경우

```
f = lambda x: 1 if x <= 1 else x * f(x - 1)
```

```
import numpy as np
```

```
def calculate_mean(arr):  
    sum = 0  
    for i in range(len(arr)):  
        sum += arr[i]  
    mean = sum / len(arr)  
    return mean
```

```
arr = [1, 2, 3, 4, 5]  
print(calculate_mean(arr))
```

코드 작성 원칙 - **KISS**를 적용한 경우

```
def faculty(number: int) -> int:
    if number <= 1:
        return 1
    else:
        return number * faculty(number - 1)
```

```
import numpy as np

arr = [1, 2, 3, 4, 5]
print(np.mean(arr))
```

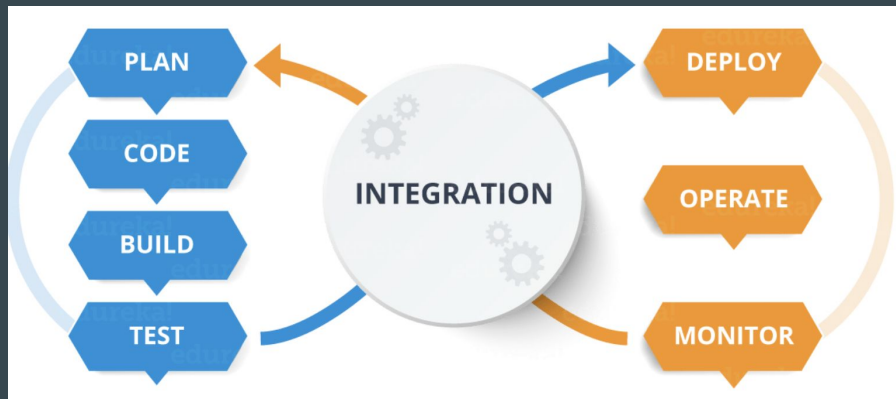
잡담: 기술 부채와 스타트업

- 스타트업의 경우 생존이 더 중요하기에 기술 부채가 점점 커짐
 - 즉 의도적으로 코드의 품질 보다는 속도에 초점을 맞추게 됨
 - 스타트업 코드에 기술 부채가 없다면 뭔가 잘못된 것임
- 하지만 어느 시점에는 이를 갚기 위해 코드/서비스 리팩토링을 시작해야함
 - 그러면 어떤 시점이 기술부채를 갚을 수 있는 시점일까?
 - 어느 정도 생존하게 되었는데 이제는 기술부채로 인해 회사의 운명이 위험해지는 경우
 - 서비스 관련 사고들의 빈도수와 심각성을 트래킹 -> 대형사고 발생 -> 기술부채를 갚아야 하는 시점
 - 테스트 작성 노력 -> 테스트 범위 (Test Coverage) 늘리기
 - CI/CD 도입
 - 새 기능 개발과 기존 코드 리팩토링을 적절히 밸런스 (예를 들면 60:40)

소프트웨어에 새로운 기능을 빠르게 출시하려면?

- 기능이 구현되는대로 소프트웨어를 바로 배포
 - vs. 정기적인 배포 (매달 혹은 매주)
- 이게 되려면 자동화된 프로세스가 만들어져야함
 - 관련 코드 커밋
 - 테스트 수행
 - 패키지 생성 ("빌드")
 - 패키지 배포
 - 모니터

이 프로세스를 CI/CD라고 부르며 보통 DevOps팀이 담당



코딩 규칙과 모범 사례 따르기

일반적인 원칙

- 일관된 포맷 및 이름 지정 규칙: `user_id` vs. `userId`, ...
- 의미 있고 설명적인 변수 이름 사용
- 적절한 주석 및 문서화
- 제어 구조와 알고리즘의 효율적인 사용

구글의 파이썬 스타일링 가이드를 중심으로 이어서 설명: BE CONSISTENT

일관된 포맷 및 이름 명명 규칙 (1)

- 개인적으로 혹은 팀 내에서 이름 명명 규칙을 정하고 진행
- 함수 이름, 변수 이름, 파일 이름은 이해하기 쉽고 약어를 피하는 것이 좋음
 - module_name, package_name, method_name, function_name
 - instance_var_name, function_parameter_name, local_var_name, global_var_name
 - ClassName, ExceptionName
 - GLOBAL_CONSTANT_NAME
 - query_proper_noun_for_thing, send_acronym_via_https

일관된 포맷 및 이름 명명 규칙 (2)

Type	Public	Internal
Packages	<code>lower_with_under</code>	
Modules	<code>lower_with_under</code>	<code>_lower_with_under</code>
Classes	<code>CapWords</code>	<code>_CapWords</code>
Exceptions	<code>CapWords</code>	
Functions	<code>lower_with_under()</code>	<code>_lower_with_under()</code>
Global/Class Constants	<code>CAPS_WITH_UNDER</code>	<code>_CAPS_WITH_UNDER</code>
Global/Class Variables	<code>lower_with_under</code>	<code>_lower_with_under</code>
Instance Variables	<code>lower_with_under</code>	<code>_lower_with_under</code> (protected)
Method Names	<code>lower_with_under()</code>	<code>_lower_with_under()</code> (protected)
Function/Method Parameters	<code>lower_with_under</code>	
Local Variables	<code>lower_with_under</code>	

일관된 포맷 및 이름 명명 규칙의 예외 (1)

- Single character names, except for specifically allowed cases:
 - counters or iterators (e.g. i, j, k, v, et al.)
 - e as an exception identifier in try/except statements.
 - f as a file handle in with statements
 - Generally speaking, descriptiveness should be proportional to the name's scope of visibility. For example, i might be a fine name for 5-line code block but within multiple nested scopes, it is likely too vague.
- Dashes (-) in any package/module name
- `__double_leading_and_trailing_underscore__` names (reserved by Python)
- Offensive terms
- Names that needlessly include the type of the variable (for example: `id_to_name_dict`)

일관된 포맷 및 이름 명명 규칙의 예외 (2)

```
try:  
    conversion_rate = paid_visitors / total_visitors  
except (ZeroDivisionError, TypeError) as e:  
    print("An error occurred: ", e)
```

```
for i in range(1, 11):  
    for j in range(1, 11):  
        print(i * j, end='\t')
```

일관된 포맷 및 이름 명명 예제


- num_items - the number of items
- is_valid - something is valid or not
- user_name - a user's name
- total_cost - the total cost of something
- max_attempts - the maximum number of attempts
- has_permission - a user has permission to do something
- error_message - an error message
- names - a list of names
- is_active - something is currently active
- input_text - an input value

ChatGPT를 이용한 변수 이름 짓기

Google Python Styling Guide를 따라서 남성 고객들로부터의 총 매출액에 해당하는 변수 이름을 짓고 싶어.

Google Python Style Guide에 따르면 변수 이름은 소문자로 작성하고 필요한 경우 밑줄 문자(_)로 단어를 구분합니다.

따라서, 여러분이 원하는 변수 이름은 다음과 같이 작성할 수 있습니다:

 Copy code

```
total_sales_by_male_customers
```

위 변수 이름은 소문자로 작성되고, 밑줄 문자를 사용하여 단어를 구분한 것입니다. 이 변수 이름은 여러 단어로 이루어져 있기 때문에 가독성이 높고 변수가 의미하는 바를 잘 전달할 수 있습니다.

적절한 주석 및 문서화

- 의미있는 주석 달기
 - 의미없는 주석의 예) `i = i + 1` # i에 1 더하기
- Example Google Style Python Docstrings

```
def function_with_pep484_type_annotations(param1: int, param2: str) -> bool:  
    """Example function with PEP 484 type annotations.
```

Parameters:

param1 (int): The first parameter.

param2 (str): The second parameter.

Returns:

value (bool): True for success, False otherwise.

```
.....
```

고급 문법을 이해하되 필요할 때만 사용하기


- Generator 함수
- List Comprehension
- Lambda 함수
- ...

코드 리뷰하기

코드 리뷰


- 주니어 개발자나 새로 온 개발자들을 트레이닝시키는 최선의 방법
- 단점은 리뷰를 해야하는 사람들이 이미 바쁜 사람들이라는 점
 - 스프린트 플래닝시 이를 고려하여 태스크 할당

5 REASONS GOOGLERS REVIEW CODE




Education

Mentoring, learning, knowledge dissemination.




Accident prevention

Find bugs and defects, ensure high quality code




Gatekeeping

Prevent arbitrary code to be committed, security



Tracing & tracking

Understanding evolution and why and how code changed



Readable Code

Maintaining norms, consistent style and design, and having adequate tests

[LEARN MORE AT WWW.MICHAELAGREILER.COM](http://www.michaelagreiler.com)

Reasons for code reviews at Google

좋은 코드 리뷰 방법

- 코드 리뷰를 요청하는 이
 - 요청시 되도록이면 조금씩 자주 요청. **Unit test**와 같이 요청하면 최상
 - 주석을 최대한 추가하고 무슨 이유에서 뭘 하려고 하는 것인지 설명
 - 리뷰에 대한 피드백을 너무 감정적으로 받아들이지 않기
- 코드 리뷰를 하는 이
 - 코딩 스타일에 대한 것보다는 코드 자체에 대해 이야기
 - 객관적으로 쓰고 비판하는 어조는 피하기
 - 충분히 시간을 들여 도움이 되는 리뷰를 제공
 - 코드 일부 리뷰하고 피드백 주지 말고 가능하면 모두 리뷰하고 전체적으로 피드백 주기
- 코드 리뷰에 편리한 툴 사용

구글 코드 리뷰 사례

- Company-wide policies for Approval
 - 1 reviewer is fine
 - Readability is important
 - Follow the coding styles of the original code (owner)
- Requirements for reviewers aren't focused on seniority or status
 - How to get a readability certification
 - What it takes to get code approved
- Code reviews are lightweight and fast
 - Requiring only one reviewer speeds up code review times
 - Small change sizes are essential for fast and high-quality code reviews

코드 리뷰 예 (1)

```
def compute_average(numbers):  
    sum = 0  
    for number in numbers:  
        sum += number  
    return sum/len(numbers)
```

- numbers가 []인 경우 divide by zero 에러 발생
- numbers가 리스트가 아닌 경우 에러 발생
- sum이란 이름은 함수 sum과 충돌 가능성. total로 대신

코드 리뷰 예 (2)

- Before: <https://github.com/keeyong/2021-code-smells/blob/main/before.py>
 - 잘못된 선택된 타입
 - 중복된 코드 (find_managers, find_vice_presidents, find_interns)
 - 내장 함수를 사용하지 않는 것 (find 함수에서 루프를 list comprehension으로 변경)
 - 모호한 변수 이름 (amount)
 - 한 함수가 굉장히 다른 2개의 일을 하게 하는 것
 - 예외처리 코드에서 아무 것도 하지 않는 것
- After: <https://github.com/keeyong/2021-code-smells/blob/main/after.py>

코드 테스트 (Unit Test)

어떤 테스트들이 존재하는가?

- **Unit Test**
 - 모듈의 특정 기능 (함수) 테스트. 보통 하나의 함수를 테스트
- **Integration Test**
 - 여러 모듈을 통합하여 하는 한 차원 위의 테스트
- **Acceptance Test**
 - 트래픽등을 생성하여 시스템에 로드를 주고 견디는지 보는 테스트
- **UI Test**
 - 요즘은 Selenium등의 툴을 이용해서 웹페이지 자체의 기능을 테스트하는 것이 대세

테스트의 중요성

- 많은 회사들이 코드 변경의 일부로 **Unit Test**를 의무적으로 요구
 - 테스트가 없으면 아예 코드 체크인이 실패
- 테스트가 많을수록 이점이 존재
 - 시스템의 안정성 증대
 - 나중에 **Refactoring**할 경우 혹은 새로 들어온 신입 엔지니어가 코드 수정할때 굉장히 편리
- 어떤 경우에는 테스트를 작성하기가 너무 힘든 경우들이 있음
 - 이 경우 스프린트 플랜닝때 시간을 넉넉히 배당

Unit Test 정의

- 자신이 만든 소프트웨어의 특정 기능을 테스트하는 것
 - 많은 경우 특정 함수를 테스트하는 것
 - 특정 입력에 대해 예상되는 특정 출력이 나오면 성공 아니면 실패 혹은 나오지 말아야할 출력이 나오면 실패하는 식으로 작성
- Integration Test나 Functional Test에 비해 가장 낮은 레벨의 **기본** 테스트

Test Coverage (1)

- 실행 가능 경로 중 몇 퍼센트나 테스트가 되어있는지 그 퍼센트를 나타낸다
- **Test Coverage**가 높을수록 시스템이 안정됨은 물론이고 부수적인 효과가 존재
 - 해당 모듈을 재작성(**Re-factoring**)시 굉장히 유용
 - 새 엔지니어가 들어와서 작업시 또한 유용

Test Coverage (2)

```
def do_something(number):
```

```
    if number < 10:
```

```
        ...
```

```
    elif number > 20:
```

```
        ...
```

```
    else:
```

```
        ...
```

100% Test Coverage를 갖기 위해서는
적어도 3개의 Unit Test가 필요 (if
조건을 모두 테스트하기 위해서)

Python Unit Test - unittest

```
# unittest 모듈을 임포트  
import unittest
```

```
# 테스트 클래스는 unittest.TestCase로부터 상속
```

```
class myTest(unittest.TestCase):
```

```
    # 테스트 환경준비가 필요하다면 setup 메소드 정의  
    def setUp(self):
```

```
        ...
```

```
    # 실제 테스트는 test_로 시작하는 메소드들로 정의
```

```
    def test_input_same_character(self):
```

```
        ...
```

```
# 테스트 스크립트를 실행하면 test_*로 시작하는 모든 함수들이 자동으로 실행됨
```

```
def test_XYZ(self):
```

```
    ...
```

Python Unit Test - unittest

- **test**로 시작하는 함수들은 각기 하나의 기능을 테스트
 - 하나의 메소드에 대해 정해진 입력을 주고 정해진 출력이 나오는지 확인
- 출력의 타입에 따라 다른 함수들을 사용해서 테스트 성공여부를 결정
 - `assertEqual(a, b)` # a와 b 두 개의 객체가 같으면 **True** (성공), 다르면 **False** (실패)
 - `assertNotEqual(a, b)` # a와 b 두 개의 객체가 다르면 **True** (성공), 같으면 **False** (실패)
 - `assertTrue(a)` # a가 **True**이면 성공, **False**이면 실패
 - `assertFalse(a)` # a가 **False**이면 성공, **True**이면 실패
 - `assertIn/assertNotIn(a, list)` # a가 list에 존재하면 전자는 성공, 존재하지 않으면 후자는 성공
 - ...

Python에서 Unit Test - Unit Test 실행

`python3 test.py` # 내 테스트 스크립트가 `test.py`인 경우

- 앞서 `unittest` 모듈을 설명했지만 `pytest`가 요즘 더 많이 쓰이고 있음
 - 클래스 없이 사용 가능하기에 좀더 간결

Python Unit Test - pytest

```
import pytest
```

```
import avg
```

```
def test_average():
```

```
    answer = avg.compute_average([1, 2, 3, 4, 5])
```

```
    assert answer == 3.0
```

```
def test_empty_input_average():
```

```
    answer = avg.compute_average([])
```

```
    assert answer is None
```

Build(Package)란?

- 자신(혹은 팀)이 개발한 소프트웨어를 최종 배포하기 위한 형태로 만드는 것을 의미
 - 많은 경우 **Docker Image**로 만들게 됨
- 이를 위해서는 충분한 테스트가 필요
 - Unit Test
 - Integration Test
 - Acceptance Test
 - UI Test

코드 테스트 (Unit Test) 실습

간단한 예제 - 숫자를 받아서 평균 계산 (avg.py)

```
import sys
```

```
def compute_average(numbers):
```

```
    if not numbers:
```

```
        return None
```

```
    total = 0
```

```
    for number in numbers:
```

```
        total += number
```

```
    return total/len(numbers)
```

```
def main():
```

```
    list = []
```

```
    for argv in sys.argv[1:]:
```

```
        list.append(int(argv))
```

```
    print("Average: " +
```

```
        str(compute_average(list)))
```

```
if __name__ == "__main__":
```

```
    main()
```

unittest 테스트 코드 작성 (test_unittest.py)

```
import unittest
```

```
import avg
```

```
class averageTestCase(unittest.TestCase):
```

```
    def test_average(self):
```

```
        answer = avg.compute_average([1, 2, 3, 4, 5])
```

```
        self.assertEqual(answer, 3.0)
```

```
    def test_empty_input_average(self):
```

```
        answer = avg.compute_average([])
```

```
        self.assertEqual(answer, None)
```

```
if __name__ == "__main__":
```

```
    unittest.main()
```

pytest 테스트 코드 작성 (test_pytest.py)

```
import pytest
```

```
import avg
```

```
def test_average():
```

```
    answer = avg.compute_average([1, 2, 3, 4, 5])
```

```
    assert answer == 3.0
```

```
def test_empty_input_average():
```

```
    answer = avg.compute_average([])
```

```
    assert answer is None
```

전체 데모

```
git clone https://github.com/keeyong/python\_test.git  
cd python_test  
python3 test_unittest.py  
pytest test_pytest.py
```


추천 자료

읽어보면 좋은 책들

- "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin
- "Code Complete: A Practical Handbook of Software Construction" by Steve McConnell
- "The Pragmatic Programmer: From Journeyman to Master" by Andrew Hunt and David Thomas
- "Refactoring: Improving the Design of Existing Code" by Martin Fowler
- "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

읽어보면 좋은 온라인 자료

- Writing clean and readable code:
 - <https://www.freecodecamp.org/news/writing-clean-code-tips-and-tricks-for-new-developers/>
- Consistent formatting and naming conventions:
 - <https://google.github.io/styleguide/>
- Using meaningful and descriptive variable names:
 - <https://www.thoughtco.com/why-variable-names-are-important-373316>
- Proper commenting and documentation:
 - <https://www.codecademy.com/articles/how-to-document-your-code>
- Writing effective test cases:
 - <https://www.guru99.com/effective-test-cases.html>
- Understanding Git and version control:
 - <https://www.atlassian.com/git/tutorials/what-is-git>

Q & A