Euler 1 and 12

Process

We started and had no idea how to mark the entries

We came up with a plan, compared it to the entries, adjusted the plan until the results felt nice

We will try and use the same structure / same priorities on future entries

Goal

Maintainable Tested Working Code

And to enjoy making it!

How to score against the goals

- Does it work? - 1 point
- Readability - 3 points
  - Names
  - Structure
- How well the code can handle change - 2 points
  - Single Responsibility
- Tests are just 1 point in round 1. Will be 3 points for future rounds
- Deep Source - 1 point

- Because of the problem domain: how performant are they is relevant… so we include that in the ‘does it work’

Goal1: Does it work?

- Without tests it is difficult to know if the code works.
- Manually testing is very hard
- For Euler1 we can ‘inspect the code’ but for later problems that will be very hard

Testing your code with Automated Tests is very helpful

And in the README file put your final answer

Goal1: Does it work?

We give 1 point out of 10 for getting the correct answer

If it calculates the answer slowly it only gets ½ a point

Basically you are assumed to get the code correct and performance is not as important as maintainability

Goal2: Readability

- Subjective
  - With naming it is hard to be objective
    - We like loop variables / lambda parameter names ‘i’ and ‘x’ etc
    - We like parameters to have ‘good’ names
    - We like methods to have ‘good names’
- Objective
  - Is there a README?
  - Comments are almost always bad.
  - Are you consistent in your layout: there are automated tools that ‘auto-reformat’ your code
  - List comprehensions are better than hand created loops

Goal2: Readability / List Comprehensions

Consider the following code

```
i=0
sum=0
while i<1000:
   if i%3==0 or i%5==0:
       sum+=i
   i+=1
print(sum)
```

In order to understand it you have to pretend to be a computer. In your head you reverse engineer what the code does.

```
print(sum([x for x in range(1000) if x % 3 == 0 or x % 5 == 0]))
```

List Comprehension

For Round 1 we did not give points for this
In Round 2+ it will probably work out about 0.5 pts

```
sum([x for x in range(1000) if x % 3 == 0 or x % 5 == 0])
```

This is an example of ‘declarative programming’. We give the logic of the computation without describing the control flow.

Declarative programming has been shown to reduce bugs, reduce lines of code, reduce ‘cognitive load’, and have a lower maintenance cost.

In general we want to encourage people to write code that has less bugs, and is cheaper to maintain.

Goal2: Readability / List Comprehensions

- Don’t ‘roll your own loops’.
  - No while true / break
  - No i = 0; while i<10 : i++
  - For loops are OK but not as good as list comprehension
- Prefer declarative code to imperative code
- Learn to love List Comprehensions

https://tylermcginnis.com/imperative-vs-declarative-programming/

https://www.pythonforbeginners.com/basics/list-comprehensions-in-python

Goal2: Readability / Comments

Comments are a code smell

The proper use of comments is to compensate
for our failure to express ourself in code.

Robert Martin

Code smells https://blog.codinghorror.com/code-smells/
Nice summary https://pragtob.wordpress.com/2017/11/14/are-comments-a-code-smell-yes-no-it-depends/
Discussion on the topic: https://softwareengineering.stackexchange.com/questions/1/comments-are-a-code-smell

More on comments

There are four types of comments

- API documentation:
  These are high maintenance, high cost comments, usually hard to read and **should** be provided if you are making an SDK/API. (But not really for this competition)
- How the code works:
  This is the worst kind of comment**. If you see them: reject the pull request**. Tell the person to write the code better so it doesn't need a comment. If the algorithm is complex, make a README
- What the code does:
  This is what tests are for. Please write your tests so that they document 'what the code does'. If you are tempted to write a comment, write a test instead
- Why you did something:
  This comments are sometimes OK. 'I used this algorithm because…' 'there is a bug and I had to a workaround' 'I really needed to tune the performance'. The last of course is suspect, so reject the pull request unless the person really needed to tune the performance

Goal 3: How well can the code handle change

- Almost all of IT is about ‘handling change well’.
- Almost all IT ‘quality rules’ are handling the need for change.

How do we design for this?

- SOLID Design principles
- YAGNI (You Ain’t Gonna Need It)

https://stackify.com/solid-design-principles/
https://www.martinfowler.com/bliki/Yagni.html

Goal 3: How well can the code handle change: SOLID

- SOLID is great for OO.
- If you squint, you get a lot of benefit out of it from Functional Programming

Single Responsibility	Easy to remember. Easily the most important principle. A piece of software should do one and only one thing
Open/Closed	Open for extension, Closed for modification I remember it as ‘program against interfaces’.
Liskov substitution principle	Complicated. I remember it as ‘do inheritance properly’
Interface Segregation	Have lots of little interfaces, rather than one big one
Dependency Inversion	Depend upon abstractions

Goal 3: How well can the code handle change: SOLID

- SOLID is great for OO.
- If you [illegible] a lot of benefit out of it from Functional Programming

sponsibility	y to remember. Easily the most important principle. A piece of are should do one and only one thing
Open/C	or exte[illegible] nber it as ‘program against interfaces’.
iskov substitutio	cated. I remember it as ‘do inheritance properly’
ace Segregation	e lots of little interfaces, rather than one big one
	Depend upon abstractions

The Euler projects are very simple
Let’s keep this simple

Goal 3: How well can the code handle change:

SINGLE RESPONSIBILITY

Loop over all the numbers and select the ones that meet the criteria

So three responsibilities

- Loop
- Check if they meet the criteria
- Sum the resulting numbers

Make a set of the multipliers of 3, another set of the multipliers of 5 and then combine them

Example way of splitting

- Make a set that is a multiplier of an integer
- Combine sets
- Sum the resulting numbers

X3 = how many multiples of 3
X5 = how many multiples of 5
x15= how many multiples of 15

Result is x3+x5-x15

Two responsibilities

- Calculate multiples
- Do the calculation

More than one responsibility in a method is the root of most evil

Goal 3: How well can the code handle change?

For Euler 1 the following are obvious changes that could happen

- The range of the data
  - Numbers between 10 and 100
- The multipliers
  - Just numbers that are multiples of 3
  - Just numbers that are multiples of 3, 7
  - The multipliers 3,5,7,11

Note that by separating the responsibilities we can handle these changes easily.

Note also that if each responsibility is separately tested, we don’t have to rewrite all the tests when we make a change to just one responsibility

Goal 4: How well tested is it?

- Has the ‘example’ integration test been checked
  - Each Euler problem comes with an example
- Has each separate responsibility been tested?

- Without automated tests we don’t know if the code works.
- We cannot safely change the code
- 3/10 points are for tests, although only 1/8 for round 1

Bonus point

- Have you set up a pipeline?

Goal 5: How performant is it?

Premature optimisation is EVIL.

- It is an abomination that should be purged from your code.
- Optimisation introduces subtle and hard to find bugs
- The rules for optimisation change every few years
  - They change depending on the libraries used, the language used, the CPU used...
- Optimisation makes code hard to understand and hard to maintain

Simple things that are good and are not premature:

- Consider the ‘order of’ rule, when selecting approach
- Consider memory usage
- Don’t introduce the N+1 problem (Look it up if you don’t recognise the name)
- Use list comprehensions instead of ‘do your own’ as these are highly optimised

Performance links

https://stackify.com/premature-optimization-evil/

https://softwareengineering.stackexchange.com/questions/80084/is-premature-optimization-really-the-root-of-all-evil

https://restfulapi.net/rest-api-n-1-problem/

https://www.infoq.com/articles/N-Plus-1/

Euler 1: points out of 8

Evaluate Code

- Does it run: 1 point (Performance is considered here)
- Are the names ‘ok’ (subjective): 1 points
- Does it look consistently linted, structured nicely: 1 point
- Uses language libraries / list comprehensions: 1 point
- Single Responsibility
  - All code in one place - 0
  - Loops separated from business logic -1
  - Ability to easy change the acceptance criteria -1
- Deep Source - 1

Evaluate Tests

Tests are just 1 point in round 1. Will be 3 points for future rounds

- Do we test the acceptance logic (e.g. divides by 3 or 5) - 1
- Do we test the looping logic - 1
- Do they use the example test? - 1

Packaging (These are bonus points and could get you 12/10)

- README or similar - 1
- A pipeline - 1

Euler 12: points out of 1**8**

Evaluate Code

- Does it run: 1 point (Performance is considered here)
- Are the names ‘ok’ (subjective): 1 points
- Does it look consistently linted, structured nicely: 1 point
- Use language libraries / list comprehensions: 1 point
- Single Responsibility/Handling Change - 2 points
- Deep source - 1 point

Evaluate Tests

Tests are just 1 point in round 1. Will be 3 points for future rounds

- Each responsibility tested - 2
- Do they use the example test? - 1

Packaging (These are bonus points and could get you 12/10)

- README or similar - 1
- A pipeline - 1