Repetition & Reinforcement

Dr Borowczak & Andey Robins

February 14-16, 2023

Quest Schedules

Week	Class	Available
3	No	Quest 1 - Take 1
4	Yes	Nothing
5	Yes	Quest 1 - Take 2
6	No	Quest 2 - Take 1
7	Yes	Nothing
8	Yes	Quest 1 - Take 3 and Quest 2 - Take 2

Repetition

Outline

- Why do we need to repeat things?
- What tools can we use?
- What do they look like in python?

Motivation

Consider this code snippet from Lab 3.

```
n = 6

if is_even(n):
  n = even_collatz(n)
else:
  n = odd_collatz(n)
```

We want to be able to repeat certain segments of code.

Just like we wanted to be able to *encapsulate* code in functions to re-use it, it makes sense to want to be able to re-use sections of code.

Instead of:

```
countdown(3)
countdown(2)
countdown(1)
countdown(0)
```

It would be nice to be able to only call `countdown()` once.

While Loops

```
n = 3
while n >= 0:
  countdown(n)
  n = n - 1
```

The above code is equivalent to the code on the previous slide.
While the change when n = 3 doesn't reduce the number of lines of
code, you can imagine how useful this would be if n = 100 or n =
857_293

Python Syntax

```
while <BOOLEAN EXPRESSION> :
  statement_1
  ...
  statement_n

more_statements
```

Lab code

Now, we'll convert this code to use a `while` loop.

```
n = 6

if n % 2 == 0:
  n = even_collatz(n)
else:
  n = odd_collatz(n)
```

Add a layer of indentation

```
n = 6

  if n % 2 == 0:
    n = even_collatz(n)
  else:
    n = odd_collatz(n)
```

Add the while keyword and a colon

```
n = 6

while ???:
  if n % 2 == 0:
    n = even_collatz(n)
  else:
    n = odd_collatz(n)
```

Add a **loop condition**

```
n = 6

while n != 1 :
  if n % 2 == 0:
    n = even_collatz(n)
  else:
    n = odd_collatz(n)
```

If you return to the lab assignment from last week, you'll notice this is exactly what loop structure is used in that code.

The `while` statement allows us to do something over and over again, no matter how many steps it takes.

What is the output of this code?

```python
n = 10
factorial = 1

while n > 1:
    factorial = factorial * n

print(factorial)
```

Infinite Loops

When using a `while` loop, it's critical that the **loop condition** updates each time through the loop.

If we run the code from the previous slide, it will never finish. The value of the `factorial` variable will never be printed.

We can understand this if we step through a few loops manually.

When a loop never exits, we call it an *infinite loop*.

Infinite loops are problematic because they prevent your program from doing what it's supposed to do.

How do we fix it?

```
n = 10
factorial = 1

while n > 1:
  factorial = factorial * n
  n = n - 1

print(factorial)
```

Updating Variables

Update Syntax

We've now seen the following structure in a few places. Let's look at it a little more in depth.

```
factorial = factorial * n
```

For those with a math background, this likely looks very wrong; however, this is perfectly legal code!

Variables Refresher

The syntax for a variable is defined as such:

```
variable_name <assignment operator> variable_value
```

And the = character is the **assignment operator** so more clearly we could write the syntax as:

```
variable_name = variable_value
```

With this in mind:

```
factorial = factorial * n
```

could be read as:

"Compute the value of `factorial` times `n`. Then assign that value to the variable `factorial`."

As an example, if `factorial` holds the value 10 and `n = 9`, running the line would look something like this:

```
factorial = factorial * n
  ->
factorial = 10 * 9
  ->
factorial = 90
```

Then, if we were to call `print(factorial)` it would print the *updated value* of 90.

An Easier Way

We've written foo = foo + 1 or foo = foo * bar a few times now. And thankfully there's an easier way!

The following are all equivalent.

```python
# use a temporary variable
tmp = factorial * n
factorial = tmp

# use the extended form we've been using
factorial = factorial * n

# use the update operator
factorial *= n
```

Update Operators

The update operators are a shortened way to take a value and update it by applying a specific operation.

Operator	Use	Expanded Form
+=	count += 1	count = count + 1
-=	count -= 1	count = count - 1
*=	factorial *= n	factorial = factorial * n
/=	half /= 2	half = half / 2
**=	n **= pow	n = n ** pow

I'll begin using these operators for clarity, you may continue using the expanded form if that makes more sense to you.

Questions?

Problem Solving with Python

What's the end goal?

It's great that we've been building up the skills to read through code and perform meaningful modifications and additions. That's one of the most common tasks in programming jobs, and likely a skill that will continue to be valuable throughout your entire computer science career.

However, another equally important skill is being able to start a problem from absolutely nothing, break it down, and solve it completely.

Encapsulation

Encapsulation: the smallest description of a single feature.

```python
def cubed(x):
    return x ** 3
```

Problem solving steps

1. Define the problem space
2. Break up the problem into individual pieces
3. Repeat until each piece seems trivial
4. Combine the pieces until you arrive at the complete solution

Where to find problems

There are a number of great online places where you can find interesting programming problems. We'll be using one of my favorites today:
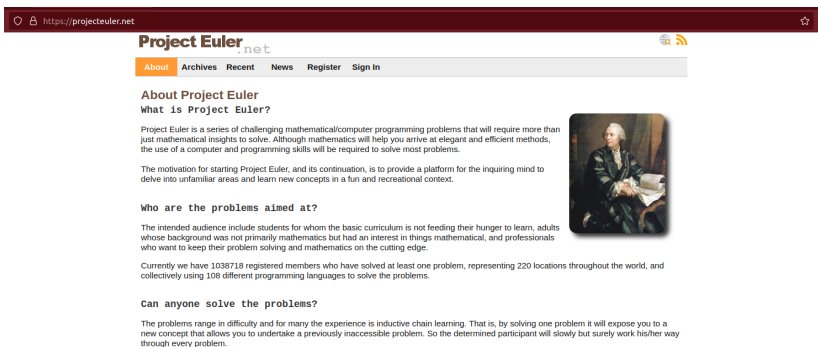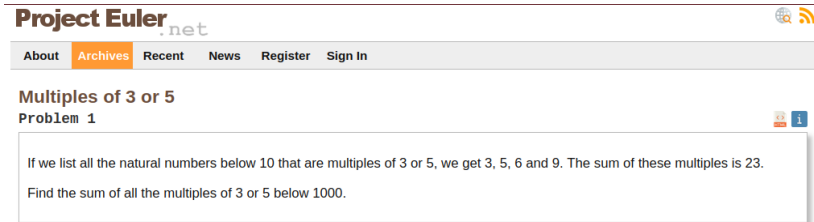
projecteuler.net

Project Euler.net

About | Archives | Recent | News | Register | Sign In

About Project Euler

What is Project Euler?

Project Euler is a series of challenging mathematical/computer programming problems that will require more than just mathematical insights to solve. Although mathematics will help you arrive at elegant and efficient methods, the use of a computer and programming skills will be required to solve most problems.

The motivation for starting Project Euler, and its continuation, is to provide a platform for the inquiring mind to delve into unfamiliar areas and learn new concepts in a fun and recreational context.

Who are the problems aimed at?

The intended audience include students for whom the basic curriculum is not feeding their hunger to learn, adults whose background was not primarily mathematics but had an interest in things mathematical, and professionals who want to keep their problem solving and mathematics on the cutting edge.

Currently we have 1038718 registered members who have solved at least one problem, representing 220 locations throughout the world, and collectively using 108 different programming languages to solve the problems.

Can anyone solve the problems?

The problems range in difficulty and for many the experience is inductive chain learning. That is, by solving one problem it will expose you to a new concept that allows you to undertake a previously inaccessible problem. So the determined participant will slowly but surely work his/her way through every problem.

Figure 1: The Project Euler home page

Problem 1

Figure 2: Today's problem

Breaking it up

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6, and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

In this description, we have both an example and the problem itself.

Find the sum of all the multiples of 3 or 5 below 1000

- ▶ Check if a number is a multiple of 3
- ▶ Check if a number is a multiple of 5
- ▶ Sum values
- ▶ Repeat the process for everything less than 1_000

Then break these down into sub steps or identify which structures we need:

- ▶ Check if a number is a multiple of 3
  - ▶ Check if the remainder of division by 3 is 0
- ▶ Check if a number is a multiple of 5
  - ▶ Check if the remainder of division by 5 is 0
- ▶ Sum values
  - ▶ + operator
- ▶ Repeat the process for everything less than 1_000
  - ▶ `while` loop

- Check if a number is a multiple of 3
  - Check if the remainder of division by 3 is 0
    - `if` and `%` modulus
- Check if a number is a multiple of 5
  - Check if the remainder of division by 5 is 0
    - `if` and `%` modulus
- Sum values
  - `+` operator
- Repeat the process for everything less than 1_000
  - `while` loop

Check if multiple of 3

We can check the multiplicity of the number by 3 using the if statement below.

```python
if x % 3 == 0:
  return True
else:
  return False
```

Since that if statement is the smallest piece of logic we can build, it makes sense to then put it inside of a function so that we can easily reuse that functionality in as many places as we need it.

```python
def is_multiple_three(x):
  if x % 3 == 0:
    return True
  else:
    return False
```

- ~~Check if a number is a multiple of 3~~
  - ~~Check if the remainder of division by 3 is 0~~
    - ~~`if` and `%` modulus~~
- Check if a number is a multiple of 5
  - Check if the remainder of division by 5 is 0
    - `if` and `%` modulus
- Sum values
  - `+` operator
- Repeat the process for everything less than 1_000
  - `while` loop

We then can use the same structure for our second problem.

```python
if x % 5 == 0:
    return True
else:
    return False
```

And put it in a function called is_multiple_five(x)

- ~~Check if a number is a multiple of 3~~
  - ~~Check if the remainder of division by 3 is 0~~
    - ~~`if` and `%` modulus~~
- ~~Check if a number is a multiple of 5~~
  - ~~Check if the remainder of division by 5 is 0~~
    - ~~`if` and `%` modulus~~
- Sum values
  - `+` operator
- Repeat the process for everything less than 1_000
  - `while` loop

The other two tasks are both individual structures, so we can begin the re-building process.

Our main process starts like so:

```python
while something < 1_000:
    # do our work here
```

We need some sort of variable to keep track of which number we're on

```python
n = 1

while n < 1_000:
    # work
```

We also know we want to avoid causing an infinite loop and actually proceed through our task

```python
n = 1

while n < 1_000:
    # work
    n += 1
```

- ~~Check if a number is a multiple of 3~~
  - ~~Check if the remainder of division by 3 is 0~~
    - ~~if and % modulus~~
- ~~Check if a number is a multiple of 5~~
  - ~~Check if the remainder of division by 5 is 0~~
    - ~~if and % modulus~~
- Sum values
  - + operator
- ~~Repeat the process for everything less than 1_000~~
  - ~~while loop~~

Find the sum of all the multiples of 3 or 5 below 1000

So let's combine our two is_multiple_... functions into one that checks both for being a multiple of 3 and 5.

```python
def is_multiple_three_or_five(x):
  return is_multiple_three(x) or is_multiple_five(x)
```

Find the sum of all the multiples of 3 or 5 below 1000

We'll need to keep track of the sum somewhere

```
n = 1
multiples_sum = 0

while n < 1_000:
  # work

  n += 1
```

```
n = 1
multiples_sum = 0

while n < 1_000:
    if is_multiple_three_or_five(n):
        multiples_sum += n
    n += 1
```

Finally, we just output our result!

```python
n = 1
multiples_sum = 0

while n < 1_000:
    if is_multiple_three_or_five(n):
        multiples_sum += n
    n += 1

print(multiples_sum)
```

Verification

The number we get doesn't mean much to us, but thankfully we have an example available to us!

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6, and 9. The sum of these multiples is 23.

```
...
while n < 10:
  ...
```

python3 euler.py -> 23 which is what we expected!

Going back to 1_000 -> 233168

Multiples of 3 or 5

Problem 1

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

Answer: **233168**
Completed on Sat, 19 Jan 2019, 21:50

Figure 3: The correct solution to the problem

Problem solving steps

1. Define the problem space
2. Break up the problem into individual pieces
3. Repeat until each piece seems trivial
4. Combine the pieces until you arrive at the complete solution

Questions?

Questions?