

While and If

Learning to use `if` and `while` is an essential skill. During this discussion, focus on what we've studied in the first three lectures: `if`, `while`, assignment (`=`), comparison (`<`, `>`, `==`), and arithmetic. Please don't use features of Python that we haven't discussed in class yet, such as `for`, `range`, and lists. We'll have plenty of time for those later in the course, but now is the time to practice the use of `if` (textbook section 1.5.4) and `while` (textbook section 1.5.5).

Problem Solving

A useful approach to implementing a function is to:

1. Pick an example input and corresponding output.
2. Describe a process (in English) that computes the output from the input using simple steps.
3. Figure out what additional names you'll need to carry out this process.
4. Implement the process in code using those additional names.
5. Determine whether the implementation really works on your original example.
6. Determine whether the implementation really works on other examples. (If not, you might need to revise step 2.)

Importantly, this approach doesn't go straight from reading a question to writing code.

For example, in the `is_prime` problem on the next page, you could:

1. Pick `n` is 9 as the input and `False` as the output.
2. Here's a process: Check that 9 (`n`) is not a multiple of any integers between 1 and 9 (`n`).
3. Introduce `i` to represent each number between 1 and 9 (`n`).
4. Implement `is_prime` (you get to do this part with your group).
5. Check that `is_prime(9)` will return `False` by thinking through the execution of the code.
6. Check that `is_prime(3)` will return `True` and `is_prime(1)` will return `False`.

Q1: Is Prime?

Write a function that returns **True** if a positive integer **n** is a prime number and **False** otherwise.

A prime number **n** is a number that is not divisible by any numbers other than 1 and **n** itself. For example, 13 is prime, since it is only divisible by 1 and 13, but 14 is not, since it is divisible by 1, 2, 7, and 14.

Use the **%** operator: **x % y** returns the remainder of **x** when divided by **y**.

```
def is_prime(n):  
    """  
    >>> is_prime(2)  
    True  
    >>> is_prime(7)  
    True  
    >>> is_prime(9)  
    False  
    >>> is_prime(1) # one is not a prime number!!  
    False  
    """  
    if n == 1:  
        return False  
    k = 2  
    while k < n:  
        if n % k == 0:  
            return False  
        k += 1  
    return True
```

Q2: Fizzbuzz

Implement the classic *Fizz Buzz* sequence. The `fizzbuzz` function takes a positive integer `n` and prints out a *single line* for each integer from 1 to `n`. For each `i`:

- If `i` is divisible by both 3 and 5, print `fizzbuzz`.
- If `i` is divisible by 3 (but not 5), print `fizz`.
- If `i` is divisible by 5 (but not 3), print `buzz`.
- Otherwise, print the number `i`.

Try to make your implementation of `fizzbuzz` concise.

```
def fizzbuzz(n):
    """
    >>> result = fizzbuzz(16)
    1
    2
    fizz
    4
    buzz
    fizz
    7
    8
    fizz
    buzz
    11
    fizz
    13
    14
    fizzbuzz
    16
    >>> print(result)
    None
    """
    i = 1
    while i <= n:
        if i % 3 == 0 and i % 5 == 0:
            print('fizzbuzz')
        elif i % 3 == 0:
            print('fizz')
        elif i % 5 == 0:
            print('buzz')
        else:
            print(i)
        i += 1
```

Q3: Ordered Digits

Implement the function `ordered_digits`, which takes as input a positive integer and returns `True` if its digits, read left to right, are in non-decreasing order, and `False` otherwise. For example, the digits of 5, 11, 127, 1357 are ordered, but not those of 21 or 1375.

```
def ordered_digits(x):
    """Return True if the (base 10) digits of X>0 are in non-decreasing
    order, and False otherwise.

    >>> ordered_digits(5)
    True
    >>> ordered_digits(11)
    True
    >>> ordered_digits(127)
    True
    >>> ordered_digits(1357)
    True
    >>> ordered_digits(21)
    False
    >>> result = ordered_digits(1375) # Return, don't print
    >>> result
    False
    """
    last = x % 10
    x = x // 10
    while x > 0:
        if last < x % 10:
            return False
        last = x % 10
        x = x // 10
    return True
```

We split off each digit in turn from the right, comparing it to the previous digit we split off, which was the one immediately to its right. We stop when we run out of digits or we find an out-of-order digit.

Q4: Unique Digits

Write a function that returns the number of unique digits in a positive integer.

Hints: You can use `//` and `%` to separate a positive integer into its one's digit and the rest of its digits.

You may find it helpful to first define a function `has_digit(n, k)`, which determines whether a number `n` has digit `k`.

```

def unique_digits(n):
    """Return the number of unique digits in positive integer n.

    >>> unique_digits(8675309) # All are unique
    7
    >>> unique_digits(13173131) # 1, 3, and 7
    3
    >>> unique_digits(101) # 0 and 1
    2
    """
    unique = 0
    while n > 0:
        last = n % 10
        n = n // 10
        if not has_digit(n, last):
            unique += 1
    return unique

# Alternate solution
def unique_digits_alt(n):
    unique = 0
    i = 0
    while i < 10:
        if has_digit(n, i):
            unique += 1
        i += 1
    return unique

def has_digit(n, k):
    """Returns whether k is a digit in n.

    >>> has_digit(10, 1)
    True
    >>> has_digit(12, 7)
    False
    """
    assert k >= 0 and k < 10
    while n > 0:
        last = n % 10
        n = n // 10
        if last == k:
            return True
    return False

```

We have provided two solutions: - In one solution, we look at the current digit, and check if the rest of the number contains that digit or not. We only say it's unique if the digit doesn't exist in the rest. We do this for every digit. - In the other, we loop through the numbers 0-9 and just call `has_digit` on each one. If it returns true then we know the entire number contains that digit and we can one to our unique count.

Q5: Repeating

Definition: A positive integer n is a *repeating sequence* of positive integer m if n is written by repeating the digits of m one or more times. For example, 616161 is a repeating sequence of 61, but 61616 is not.

Implement `repeating` which takes positive integers t and n . It returns whether n is a repeating sequence of some t -digit integer.

```
def repeating(t, n):
    """Return whether t digits repeat to form positive integer n.

    >>> repeating(1, 6161)
    False
    >>> repeating(2, 6161) # repeats 61 (2 digits)
    True
    >>> repeating(3, 6161)
    False
    >>> repeating(4, 6161) # repeats 6161 (4 digits)
    True
    >>> repeating(5, 6161) # there are only 4 digits
    False
    """
    if pow(10, t-1) > n: # make sure n has at least t digits
        return False
    end = n % pow(10, t)
    rest = n
    while rest:
        if rest % pow(10, t) != end:
            return False
        rest = rest // pow(10, t)
    return True
```

The iterative process needed to implement this function is to check that the last t digits of the `rest` match the last t digits of n , then remove the last t digits of `rest`.