Simple Recursion

Rob Hackman

Fall 2025

University of Alberta

Table of Contents

Function Calls

Some important things we've learned so far, and what we're doing today

- We define functions for procedures we want to reuse, likely with different arguments
- We learned that functions can call other functions, to solve sub-problems, e.g. our tax example

Function Calls

Some important things we've learned so far, and what we're doing today

- We define functions for procedures we want to reuse, likely with different arguments
- We learned that functions can call other functions, to solve sub-problems, e.g. our tax example
- But what if the sub-problem we want to solve is another instance of the original problem?

Function Calls

Some important things we've learned so far, and what we're doing today

- We define functions for procedures we want to reuse, likely with different arguments
- We learned that functions can call other functions, to solve sub-problems, e.g. our tax example
- But what if the sub-problem we want to solve is another instance of the original problem?
    - This is where *recursion* becomes useful

Table of Contents

Defining recursion

Recursion occurs when we define something in terms of itself.

For example: “Fewer students show up to class each day of classes. The number of students that show up on any given day of classes is 95% of the students that showed up on the previous day, rounded up to the nearest natural number”

Recursive Example: Attendance

Here is a function for the number of students that show up on the nth day of classes

$$Attendance(n) = \lceil 0.95 \times Attendance(n-1) \rceil$$

Recursive Example: Attendance

Here is a function for the number of students that show up on the nth day of classes

$$Attendance(n) = \lceil 0.95 \times Attendance(n-1) \rceil$$

Can we calculate how many students show up on the fifth day of classes?

Recursive Example: Attendance

Here is a function for the number of students that show up on the nth day of classes

$$Attendance(n) = \lceil 0.95 \times Attendance(n-1) \rceil$$

Can we calculate how many students show up on the fifth day of classes?

Need to know how many students showed up on the first day!

Recursive Example: Attendance

Here is our w function, now a piecewise function

$$Attendance(n) = \begin{cases} 156 & \text{if } n = 1 \\ \lceil 0.95 \times Attendance(n-1) \rceil & \text{if } n > 1 \end{cases}$$

Now can we calculate how many students showed up on the fifth day of classes - Attendance(5)

Recursive Example: Calculating Attendance(5)

$$Attendance(5) = \lceil 0.95 \times Attendance(4) \rceil$$

Recursive Example: Calculating Attendance(5)

$$Attendance(5) = \lceil 0.95 \times Attendance(4) \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times Attendance(3) \rceil \rceil$$

Recursive Example: Calculating Attendance(5)

$$Attendance(5) = \lceil 0.95 \times Attendance(4) \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times Attendance(3) \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(2) \rceil \rceil \rceil$$

Recursive Example: Calculating Attendance(5)

$$Attendance(5) = \lceil 0.95 \times Attendance(4) \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times Attendance(3) \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(2) \rceil \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(1) \rceil \rceil \rceil \rceil$$

Recursive Example: Calculating Attendance(5)

$$Attendance(5) = \lceil 0.95 \times Attendance(4) \rceil$$
$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times Attendance(3) \rceil \rceil$$
$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(2) \rceil \rceil \rceil$$
$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(1) \rceil \rceil \rceil \rceil$$
$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times 156 \rceil \rceil \rceil \rceil$$

Recursive Example: Calculating Attendance(5)

$$Attendance(5) = \lceil 0.95 \times Attendance(4) \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times Attendance(3) \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(2) \rceil \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(1) \rceil \rceil \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times 156 \rceil \rceil \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times 149 \rceil \rceil \rceil$$

Recursive Example: Calculating Attendance(5)

$$Attendance(5) = \lceil 0.95 \times Attendance(4) \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times Attendance(3) \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(2) \rceil \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(1) \rceil \rceil \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times 156 \rceil \rceil \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times 149 \rceil \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times 142 \rceil \rceil$$

Recursive Example: Calculating Attendance(5)

$$Attendance(5) = \lceil 0.95 \times Attendance(4) \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times Attendance(3) \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(2) \rceil \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(1) \rceil \rceil \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times 156 \rceil \rceil \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times 149 \rceil \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times \lceil 0.95 \times 142 \rceil \rceil$$

$$Attendance(5) = \lceil 0.95 \times 135 \rceil$$

Recursive Example: Calculating Attendance(5)

$$
\begin{aligned}
Attendance(5) &= \lceil 0.95 \times Attendance(4) \rceil \\
Attendance(5) &= \lceil 0.95 \times \lceil 0.95 \times Attendance(3) \rceil \rceil \\
Attendance(5) &= \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(2) \rceil \rceil \rceil \\
Attendance(5) &= \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times Attendance(1) \rceil \rceil \rceil \rceil \\
Attendance(5) &= \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times 156 \rceil \rceil \rceil \rceil \\
Attendance(5) &= \lceil 0.95 \times \lceil 0.95 \times \lceil 0.95 \times 149 \rceil \rceil \rceil \\
Attendance(5) &= \lceil 0.95 \times \lceil 0.95 \times 142 \rceil \rceil \\
Attendance(5) &= \lceil 0.95 \times 135 \rceil \\
Attendance(5) &= 129
\end{aligned}
$$

What is necessary for recursion?

When writing our attendance function what did we discover?

What two things did the recursion require?

What is necessary for recursion?

When writing our attendance function what did we discover?

What two things did the recursion require?

- At least one *recursive case*
- At least one *base case*

What is necessary for recursion?

When writing our attendance function what did we discover?

What two things did the recursion require?

- At least one *recursive case*
- At least one *base case*

Note: This is true for any correctly defined recursive function

Components of Recursion

A *recursive case* is when our next calculation is defined in terms of the function itself, e.g.

$$Attendance(n) = \lceil 0.95 \times Attendance(n-1) \rceil \text{ if } n > 1$$

A *base case* is a stopping point of our recursion, when we no longer define our result in terms of further applications of the function, e.g.

$$Attendance(1) = 100 \text{ if } n = 1$$

Table of Contents

Attendance function in Python

Let's translate our mathematical attendance function into a Python function:

$$Attendance(n) = \begin{cases} 156 & \text{if } n = 1 \\ \lceil 0.95 \times Attendance(n-1) \rceil & \text{if } n > 1 \end{cases}$$

Attendance function in Python

Let’s translate our mathematical attendance function into a Python function

Attendance function in Python

Let's translate our mathematical attendance function into a Python function

```python
def attendance(n):
  '''
  attendance returns the number of students attending
  CMPUT274 on the nth day of classes

  n - a natural number greater than or equal to 1
  Returns - An integer

  Examples
  attendance(5)  -> 129
  attendance(10) -> 102
  '''
```

Attendance function in Python

Let's translate our mathematical attendance function into a Python function

```python
import math
def attendance(n):
  '''
  ...
  '''
  if n == 1:
    return 156
  else:
    return math.ceil(0.95*attendance(n-1))
```

Table of Contents

How to create a recursive solution

Creating a recursive solution is not always easy

In order to solve a problem recursively you must identify

- At least one base case
- At least one recursive case
- How to “step” closer to the base case
- How to build your result from the recursive result

It is not always obvious how to define each of these four!

Solving the `leetSpeak` function

Now we will attempt to use recursion to implement our `leetSpeak` function. Let's identify what we can use for each of these components.

- At least one base case
- At least one recursive case
- How to "step" closer to the base case
- How to build your result from the recursive result

Solving the `leetSpeak` function

Now we will attempt to use recursion to implement our `leetSpeak` function. Let's identify what we can use for each of these components.

- At least one base case
    - The empty string `""` is trivially convertible to leet speak.
- At least one recursive case
- How to "step" closer to the base case
- How to build your result from the recursive result

Solving the `leetSpeak` function

Now we will attempt to use recursion to implement our `leetSpeak` function. Let's identify what we can use for each of these components.

- At least one base case
    - The empty string `""` is trivially convertible to leet speak.
- At least one recursive case
    - Any non-empty is a case we still have work to do.
- How to "step" closer to the base case
- How to build your result from the recursive result

Solving the `leetSpeak` function

Now we will attempt to use recursion to implement our `leetSpeak` function. Let's identify what we can use for each of these components.

- At least one base case
    - The empty string `""` is trivially convertible to leet speak.
- At least one recursive case
    - Any non-empty is a case we still have work to do.
- How to "step" closer to the base case
    - The easiest thing to do is to work one character at a time. We can step closer to the base case by removing one character at a time. The *first* character is a good choice.
- How to build your result from the recursive result

Solving the `leetSpeak` function

Now we will attempt to use recursion to implement our `leetSpeak` function. Let's identify what we can use for each of these components.

- At least one base case
  - The empty string `""` is trivially convertible to leet speak.
- At least one recursive case
  - Any non-empty is a case we still have work to do.
- How to "step" closer to the base case
  - The easiest thing to do is to work one character at a time. We can step closer to the base case by removing one character at a time. The *first* character is a good choice.
- How to build your result from the recursive result
  - Convert the character we removed and combine it with the recursive result to produce our result.

`leetSpeak` **helper function**

```python
def convertChar(c):
  if c == "A" or c == "a":
    return "4"
  if c == "E" or c == "e":
    return "3"
  if c == "L" or c == "l":
    return "1"
  if c == "T" or c == "t":
    return "7"
  if c == "S" or c == "s":
    return "5"
  if c == "O" or c == "o":
    return "0"
  return c
```

leetSpeak implementation

```
def leetSpeak(s):
  if s == "":
    return ""
  return convertChar(s[0]) + leetSpeak(s[1:])
```

Example application of `leetSpeak`

```
leetSpeak("leet")
```

Example application of `leetSpeak`

```
leetSpeak("leet")
 = convertChar("l") + leetSpeak("eet")
```

Example application of `leetSpeak`

```
leetSpeak("leet")
 = convertChar("l") + leetSpeak("eet")
 = convertChar("l") + convertChar("e") + leetSpeak("et")
```

Example application of `leetSpeak`

```
leetSpeak("leet")
 = convertChar("l") + leetSpeak("eet")
 = convertChar("l") + convertChar("e") + leetSpeak("et")
 = convertChar("l") + convertChar("e") + convertChar("e")
   + leetSpeak("t")
```

Example application of `leetSpeak`

```
leetSpeak("leet")
 = convertChar("l") + leetSpeak("eet")
 = convertChar("l") + convertChar("e") + leetSpeak("et")
 = convertChar("l") + convertChar("e") + convertChar("e")
   + leetSpeak("t")
 = convertChar("l") + convertChar("e") + convertChar("e")
   + converChar("t") + leetSpeak("")
```

Example application of `leetSpeak`

```
leetSpeak("leet")
 = convertChar("l") + leetSpeak("eet")
 = convertChar("l") + convertChar("e") + leetSpeak("et")
 = convertChar("l") + convertChar("e") + convertChar("e")
   + leetSpeak("t")
 = convertChar("l") + convertChar("e") + convertChar("e")
   + converChar("t") + ""
```

Example application of `leetSpeak`

```
leetSpeak("leet")
 = convertChar("l") + leetSpeak("eet")
 = convertChar("l") + convertChar("e") + leetSpeak("et")
 = convertChar("l") + convertChar("e") + convertChar("e")
   + "7"
```

Example application of `leetSpeak`

```
leetSpeak("leet")
 = convertChar("l") + leetSpeak("eet")
 = convertChar("l") + convertChar("e") + "37"
```

Example application of `leetSpeak`

```
leetSpeak("leet")
 = convertChar("l") + "337"
```

Example application of `leetSpeak`

```
leetSpeak("leet")
 = "1337"
```

`leetSpeak` bug!

However, if we carefully test our `leetSpeak` function we see there is a bug.

```
def leetSpeak(s):
  if s == "":
    return ""
  return convertChar(s[0]) + leetSpeak(s[1:])
```

`leetSpeak` — words that end in `er`

One of our rules for `leetSpeak` conversion was that words that end in `er` have `er` replaced with `z0r`.
How can we fix our implementation?

```
def leetSpeak(s):
  if s == "":
    return ""
  return convertChar(s[0]) + leetSpeak(s[1:])
```

leetSpeak — words that end in er

One of our rules for leetSpeak conversion was that words that end in er have er replaced with z0r.
How can we fix our implementation?

A function does not have to have only one base case, just at least one. If our function receives a string that is exactly er that means we should produce z0r as the string ‘‘er’’ is a string that ends in er.

```
def leetSpeak(s):
  if s == "":
    return ""
  return convertChar(s[0]) + leetSpeak(s[1:])
```

leetSpeak — words that end in er

One of our rules for leetSpeak conversion was that words that end in er have er replaced with z0r.
How can we fix our implementation?

A function does not have to have only one base case, just at least one. If our function receives a string that is exactly er that means we should produce z0r as the string ‘‘er’’ is a string that ends in er.

```python
def leetSpeak(s):
  if s == "":
    return ""
  if s == "er":
    return "z0r"
  return convertChar(s[0]) + leetSpeak(s[1:])
```

Practice Question 1

Sometimes it can be beneficial for us to know the length of a string. As such, we will write a function to compute this for us.

Practice Question: Write a function `strlen` that has one string parameter and returns the length of that string.

Practice question 2

We've been using == to check equality of two strings — however this is actually a complex operation not a basic one. Determining an answer requires in the worst case comparing all the characters of two strings one-by-one.

In an exercise to understand some of the work Python must do for us when implementing string equality we will write it ourselves.

Practice Question: Write the function `streq` that has two string parameters and returns `True` if those strings the same and `False` otherwise. You may not use the == operator except between strings that are at most one character.

Recursive data definitions

Recursion on strings makes some intuitive sense, as we can easily "pick-apart" strings one character at a time to operate on them. Our recursive case was always getting one step closer to the trivial string base case (an empty string).

Recursive data definitions

Recursion on strings makes some intuitive sense, as we can easily "pick-apart" strings one character at a time to operate on them. Our recursive case was always getting one step closer to the trivial string base case (an empty string).

In fact, we can define the set of strings recursively. Let us define the data type `char` as a data type that represents any single character. Then we can define strings with the following statements.

Recursive data definitions

Recursion on strings makes some intuitive sense, as we can easily "pick-apart" strings one character at a time to operate on them. Our recursive case was always getting one step closer to the trivial string base case (an empty string).

In fact, we can define the set of strings recursively. Let us define the data type `char` as a data type that represents any single character. Then we can define strings with the following statements.

A `str` is one of:

- The empty string: `""`
- A `char` + `str`

Structural recursion

The style of recursive functions we've written operating on strings is sometimes called *structural recursion*. This is because the form of the recursive solution follows directly form the definition of the data itself.

Structural recursion is straightforward and easy to get right, as the base cases and recursive cases often come directly from the data itself.

What would structural recursion look like on the natural numbers?

Naturals defined recursively

If we want to write structural recursion on a data type, we must have a recursive definition for the structure of the data! So, how can we define the natural numbers?

Naturals defined recursively

If we want to write structural recursion on a data type, we must have a recursive definition for the structure of the data! So, how can we define the natural numbers?

A member of the natural numbers ($\mathbb{N}$) is one of:

Naturals defined recursively

If we want to write structural recursion on a data type, we must have a recursive definition for the structure of the data! So, how can we define the natural numbers?

A member of the natural numbers ($\mathbb{N}$) is one of:

- 0

Naturals defined recursively

If we want to write structural recursion on a data type, we must have a recursive definition for the structure of the data! So, how can we define the natural numbers?

A member of the natural numbers ($\mathbb{N}$) is one of:

- 0
- $1 + n,\ n \in \mathbb{N}$

Naturals defined recursively

If we want to write structural recursion on a data type, we must have a recursive definition for the structure of the data! So, how can we define the natural numbers?

A member of the natural numbers ($\mathbb{N}$) is one of:

- 0
- $1 + n,\ n \in \mathbb{N}$

Aside: Recursive data definitions like this are very common in computer science, in fact this isn't the first time we've seen a recursive definition of an entity. Hint: recursive definitions are used all the time for defining parts of a programming languages grammar!

Factorial function

Recall the factorial of a natural number n, denoted as $n!$ is defined as:

$$n! = \prod_{i=1}^{n} i$$

Any summation or repeated product can easily be written as a recursion.

Factorial function in Python

```
def factorial(n):
  if n == 0:
    return 0
  if n == 1:
    return 1
  return n*factorial(n-1)
```

This implementation of the factorial function is an example of structural recursion on natural numbers as we've defined them.

A different way to view numbers

Let's consider another problem we may want to solve in Python.

Problem: write a recursive function `digitSum` that calculates the sum of all digits in a number, e.g. `digitSum(1026)` $\longrightarrow$ `9`

Summing digits in 735

How might we sum the digits in the number 735?

Summing digits in 735

How might we sum the digits in the number 735?

7

Summing digits in 735

How might we sum the digits in the number 735?

$$7 + 3$$

Summing digits in 735

How might we sum the digits in the number 735?

$$7 + 3 + 5$$

Summing digits in 735

How might we sum the digits in the number 735?

$$7 + 3 + 5$$

This seems obvious - but what process did we actually follow in our head to achieve this?

Summing digits in 735

How might we sum the digits in the number 735?

$$7 + 3 + 5$$

This seems obvious - but what process did we actually follow in our head to achieve this?

We took the leftmost digit out and included it in our sum, and then repeated this process until there were no digits left to add to our summation.

Summing digits in 735

How might we sum the digits in the number 735?

$$7 + 3 + 5$$

This seems obvious - but what process did we actually follow in our head to achieve this?

We took the leftmost digit out and included it in our sum, and then repeated this process until there were no digits left to add to our summation.

Can we apply this intuitive solution to Python code?

Digit summing — take one

How can we translate this process into a generally stated formula?

How do we grab that first most significant digit? How do we calculate the rest of the number

Digit summing — take one MSD extraction

We need to find the mathematical function parameterized by a number x which produces the most significant digit of x

Digit summing — take one MSD extraction

We need to find the mathematical function parameterized by a number x which produces the most significant digit of x

For 735 we can achieve this with

$$MSD(735) = \lfloor \frac{735}{100} \rfloor$$

Digit summing — take one MSD extraction

We need to find the mathematical function parameterized by a number x which produces the most significant digit of x

For 735 we can achieve this with

$$\begin{aligned} MSD(735) &= \lfloor \frac{735}{100} \rfloor \\ &= \lfloor 7.35 \rfloor \end{aligned}$$

Digit summing — take one MSD extraction

We need to find the mathematical function parameterized by a number *x* which produces the most significant digit of *x*

For 735 we can achieve this with

$$\begin{aligned} MSD(735) &= \lfloor \frac{735}{100} \rfloor \\ &= \lfloor 7.35 \rfloor \\ &= 7 \end{aligned}$$

Digit summing — take one MSD extraction

We need to find the mathematical function parameterized by a number x which produces the most significant digit of x

For 735 we can achieve this with

$$\begin{aligned} MSD(735) &= \lfloor \frac{735}{100} \rfloor \\ &= \lfloor 7.35 \rfloor \\ &= 7 \end{aligned}$$

What about a 5 digit number?

Digit summing — take one MSD extraction

We need to find the mathematical function parameterized by a number x which produces the most significant digit of x

For 735 we can achieve this with

$$\begin{aligned} MSD(735) &= \lfloor \frac{735}{100} \rfloor \\ &= \lfloor 7.35 \rfloor \\ &= \ 7 \end{aligned}$$

What about a 5 digit number? 7 digit number?

Digit summing — take one MSD extraction

We need to find the mathematical function parameterized by a number x which produces the most significant digit of x

For 735 we can achieve this with

$$\begin{aligned} MSD(735) &= \lfloor \frac{735}{100} \rfloor \\ &= \lfloor 7.35 \rfloor \\ &= 7 \end{aligned}$$

What about a 5 digit number? 7 digit number? n digit number?

Digit summing - take one MSD extraction

In the calculation $MSD(735) = \lfloor \frac{735}{100} \rfloor$ the divisor 100 came from somewhere - but where?

Digit summing - take one MSD extraction

In the calculation $MSD(735) = \lfloor \frac{735}{100} \rfloor$ the divisor 100 came from somewhere - but where?

$100 = 10^2$

Digit summing - take one MSD extraction

In the calculation $MSD(735) = \lfloor \frac{735}{100} \rfloor$ the divisor 100 came from somewhere - but where?

$100 = 10^2$
$log_{10} 100 = 2$

Digit summing - take one MSD extraction

In the calculation $MSD(735) = \lfloor \frac{735}{100} \rfloor$ the divisor 100 came from somewhere - but where?

$100 = 10^2$
$log_{10} 100 = 2$
$log_{10} 735 = 2.86628...$

Digit summing - take one MSD extraction

In the calculation $MSD(735) = \lfloor \frac{735}{100} \rfloor$ the divisor 100 came from somewhere - but where?

$100 = 10^2$

$log_{10}100 = 2$

$log_{10}735 = 2.86628...$

$log_{10}999 = 2.99956...$

Digit summing - take one MSD extraction

In the calculation $MSD(735) = \lfloor \frac{735}{100} \rfloor$ the divisor 100 came from somewhere - but where?

$100 = 10^2$
$log_{10} 100 = 2$
$log_{10} 735 = 2.86628...$
$log_{10} 999 = 2.99956...$
$log_{10} 1000 = 3$

Digit summing - take one MSD extraction

In the calculation $MSD(735) = \lfloor \frac{735}{100} \rfloor$ the divisor 100 came from somewhere - but where?

$100 = 10^2$
$log_{10} 100 = 2$
$log_{10} 735 = 2.86628...$
$log_{10} 999 = 2.99956...$
$log_{10} 1000 = 3$

We can devise a function from here, but it will be tricky.

`logFloor`

Exercise: Write the function `logFloor` that takes a number greater than zero and produces the floor of the base ten log of that number.

Aside: checking preconditions

We have written several functions now that have had various requirements on their parameters. For example, the `logFloor` function requires that the given argument be greater than 0, as the log of 0 is undefined and the logic for calculating negative powers would be different.

What should be done if the *client programmer*[1] violates our functions requirements?

[1]A *client programmer* is any programmer that uses a module we've written. When we write functions a client programmer is any programmer that calls our function.

Violated preconditions option 1: garbage in, garbage out

A simple policy for when a client programmer violates preconditions is that of “garbage in, garbage out”.

Violated preconditions option 1: garbage in, garbage out

A simple policy for when a client programmer violates preconditions is that of “garbage in, garbage out”.

Simply put this policy states that should a client programmer give you data that violates your preconditions (garbage), then the value your function produces is meaningless (garbage).

Violated preconditions option 1: garbage in, garbage out

A simple policy for when a client programmer violates preconditions is that of “garbage in, garbage out”.

Simply put this policy states that should a client programmer give you data that violates your preconditions (garbage), then the value your function produces is meaningless (garbage).

In many cases this policy is just fine. Sometimes though, it can be nice to catch a mistake when it is made, rather than allow the client programmers mistake to go unnoticed right away.

Violated preconditions option 2: crash the program

If a programmer violates the preconditions of a function, it is in their best interest to be informed of that as soon as possible. If the function simply quietly produces a garbage value, the client programmer may not notice the mistake they've made right away.

Down the line the client programmer will have a bug from using the garbage value produced, and might not immediately realize it was caused by an improper call of that function they performed!

Violated preconditions option 2: crash the program

If a programmer violates the preconditions of a function, it is in their best interest to be informed of that as soon as possible. If the function simply quietly produces a garbage value, the client programmer may not notice the mistake they've made right away.

Down the line the client programmer will have a bug from using the garbage value produced, and might not immediately realize it was caused by an improper call of that function they performed!

So it may be beneficial to cause a function to fail *loudly* when a caller provides invalid arguments. One way to fail loudly is to crash the program.

`assert` statements

This introduces for us a new keyword and new type of statement — assert statements. The form of an assert statement is:

`assert` bExpr

Where `bExpr` is an expression that can be evaluated to a boolean value.

Behaviour of `assert` statements

The behaviour of an assert statement is simple.

An assert statement

- does nothing if the boolean expression evaluates to `True`
- crashes the program otherwise

Adding an assert statement to `logFloor`

If we want to fail loudly we should include `assert` statements at the beginning of each of our functions to check all of our preconditions. Let's update `logFloor` to check its preconditions

```
def logFloor(n):
  assert n > 0
  ...
```

The `digitSum` function

Now that we have the `logFloor` function can we write `digitSum`?

Exercise: Complete the `digitSum` function so that it calculates the digit sum in the way we have designed, working on the most significant digit and performing recursion on the rest of the number.

Thoughts on the `digitSum` function

The `digitSum` Function as we have written it is more difficult than it needs to be.

The intuitive way we perform things in real life, in our minds, or on paper, does not always translate well to code.

Perhaps we should rethink our approach to `digitSum`.

Digit summing - take two

Rethinking our approach, how else could we have summed the digits of 735?

Digit summing - take two

Rethinking our approach, how else could we have summed the digits of 735?

5

Digit summing - take two

Rethinking our approach, how else could we have summed the digits of 735?

$$5 + 3$$

Digit summing - take two

Rethinking our approach, how else could we have summed the digits of 735?

$$5 + 3 + 7$$

Digit summing - take two

Rethinking our approach, how else could we have summed the digits of 735?

$$5 + 3 + 7$$

Same approach as before, but now we are plucking the least significant digit rather than the most significant digit - is this helpful?

Digit summing - take two

Rethinking our approach, how else could we have summed the digits of 735?

$$5 + 3 + 7$$

Same approach as before, but now we are plucking the least significant digit rather than the most significant digit - is this helpful?

Observation: The least significant digit is always in the same location in the number making extraction easier

Modulo and integer division

There are two operators we have not discussed yet that can be particularly useful to us.

- The modulo operator, `%`, is a binary operator that returns the remainder[2] when its first operand is divided by its second operand. For example `17%5` $\longrightarrow$ `2`
- The integer division operator, `//`, returns the whole number quotient when division is performed between its two operands. For example `17//5` $\longrightarrow$ `3`

[2]When applied to positive operands.

Counting digits recursively

So how can we break down the summing the digits of a number into a recursive function?

- Base case?
- Recursive case?
- Step closer to base?
- Build result?

Counting digits recursively

So how can we break down the summing the digits of a number into a recursive function?

- Base case? A single digit number
- Recursive case?
- Step closer to base?
- Build result?

Counting digits recursively

So how can we break down the summing the digits of a number into a recursive function?

- Base case? A single digit number
- Recursive case? A number with more than one digit
- Step closer to base?
- Build result?

Counting digits recursively

So how can we break down the summing the digits of a number into a recursive function?

- Base case? A single digit number
- Recursive case? A number with more than one digit
- Step closer to base? Remove a digit, making the next recursive argument one digit fewer
- Build result?

Counting digits recursively

So how can we break down the summing the digits of a number into a recursive function?

- Base case? A single digit number
- Recursive case? A number with more than one digit
- Step closer to base? Remove a digit, making the next recursive argument one digit fewer
- Build result? Add the removed digit to the recursive result

Counting digits recursively - mathematic definition

So how can we convert these steps into a piecewise function?

Counting digits recursively - mathematic definition

So how can we convert these steps into a piecewise function?

$$digitSum(n) = \begin{cases} n & \text{if } n < 10 \\ & \end{cases}$$

Counting digits recursively - mathematic definition

So how can we convert these steps into a piecewise function?

$$digitSum(n) = \begin{cases} n & \text{if } n < 10 \\ n \bmod 10 + digitSum(\lfloor \frac{n}{10} \rfloor) & \text{if } n \geq 10 \end{cases}$$

Tracing the formula for the digits of 735

$$digitSum(735) = 735 \bmod 10 + digitSum(\lfloor \frac{735}{10} \rfloor)$$

Tracing the formula for the digits of 735

$$digitSum(735) = 735 \bmod 10 + digitSum(\lfloor \frac{735}{10} \rfloor)$$
$$digitSum(735) = 5 + digitSum(73)$$

Tracing the formula for the digits of 735

$$digitSum(735) = 735 \bmod 10 + digitSum(\lfloor \frac{735}{10} \rfloor)$$

$$digitSum(735) = 5 + digitSum(73)$$

$$digitSum(735) = 5 + 73 \bmod 10 + digitSum(\lfloor \frac{73}{10} \rfloor)$$

Tracing the formula for the digits of 735

$$\mathit{digitSum}(735) = 735 \bmod 10 + \mathit{digitSum}(\lfloor \frac{735}{10} \rfloor)$$
$$\mathit{digitSum}(735) = 5 + \mathit{digitSum}(73)$$
$$\mathit{digitSum}(735) = 5 + 73 \bmod 10 + \mathit{digitSum}(\lfloor \frac{73}{10} \rfloor)$$
$$\mathit{digitSum}(735) = 5 + 3 + \mathit{digitSum}(7)$$

Tracing the formula for the digits of 735

$$\mathit{digitSum}(735) = 735 \bmod 10 + \mathit{digitSum}(\lfloor \frac{735}{10} \rfloor)$$

$$\mathit{digitSum}(735) = 5 + \mathit{digitSum}(73)$$

$$\mathit{digitSum}(735) = 5 + 73 \bmod 10 + \mathit{digitSum}(\lfloor \frac{73}{10} \rfloor)$$

$$\mathit{digitSum}(735) = 5 + 3 + \mathit{digitSum}(7)$$

$$\mathit{digitSum}(735) = 5 + 3 + 7$$

Tracing the formula for the digits of 735

$$\mathit{digitSum}(735) = 735 \bmod 10 + \mathit{digitSum}(\lfloor \frac{735}{10} \rfloor)$$

$$\mathit{digitSum}(735) = 5 + \mathit{digitSum}(73)$$

$$\mathit{digitSum}(735) = 5 + 73 \bmod 10 + \mathit{digitSum}(\lfloor \frac{73}{10} \rfloor)$$

$$\mathit{digitSum}(735) = 5 + 3 + \mathit{digitSum}(7)$$

$$\mathit{digitSum}(735) = 5 + 3 + 7$$

$$\mathit{digitSum}(735) = 5 + 10$$

Tracing the formula for the digits of 735

$$
\begin{aligned}
digitSum(735) &= 735 \bmod 10 + digitSum(\lfloor \frac{735}{10} \rfloor) \\
digitSum(735) &= 5 + digitSum(73) \\
digitSum(735) &= 5 + 73 \bmod 10 + digitSum(\lfloor \frac{73}{10} \rfloor) \\
digitSum(735) &= 5 + 3 + digitSum(7) \\
digitSum(735) &= 5 + 3 + 7 \\
digitSum(735) &= 5 + 10 \\
digitSum(735) &= 15
\end{aligned}
$$

Counting digits recursively - Python definition

Now translate this function into Python

Counting digits recursively - Python definition

Now translate this function into Python

```
def digitSum(n):
  if n < 10:
    return n
  else:
    return n%10 + digitSum(n//10)
```

`digitSum` — structural recursion?

Is our `digitSum` function structural recursion?

`digitSum` — structural recursion?

Is our `digitSum` function structural recursion?

Not for our previous definition of the natural numbers.

`digitSum` — structural recursion?

Is our `digitSum` function structural recursion?

Not for our previous definition of the natural numbers.

But is there only one way to define the natural numbers?

Natural numbers, a second definition

We can instead define the natural number in terms of digits. Let a natural number (Nat) be one of

- A digit
- $10 * Nat$ + a digit

Where a digit is defined as one of $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

Natural numbers, a second definition

We can instead define the natural number in terms of digits. Let a natural number (Nat) be one of

- A digit
- $10 * \mathit{Nat} +$ a digit

Where a digit is defined as one of $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

Then relative to this definition our `digitSum` function is structurally recursive!