# Simple Functions in Python

Rob Hackman

Fall 2025

University of Alberta

# Table of Contents


Writing Simple Functions in Python

- Our first Python Function
- Python Statements

Function Specification and Comments

Conditional Execution

Boolean Logic

Understanding Python Conditionals

# Functions — program building blocks

Most programs are made up of many small functions composed together to solve large-scale problems.

# Functions — program building blocks

Most programs are made up of many small functions composed together to solve large-scale problems.

We will start learning Python with functions being our simple building blocks.

## Functions — program building blocks

Most programs are made up of many small functions composed together to solve large-scale problems.

We will start learning Python with functions being our simple building blocks.

As we learn Python we will introduce and use more features of the language, but to start out we will be very restricted! If you have already worked in Python be prepared to learn a different approach to writing it.

# Translating an algebraic function to Python

First, we must know how to write Python code! The first thing we will learn is that our algebraic expressions can also be written in Python, meaning the same thing. Consider the following function, and an attempted translation to Python.

# Translating an algebraic function to Python

First, we must know how to write Python code! The first thing we will learn is that our algebraic expressions can also be written in Python, meaning the same thing. Consider the following function, and an attempted translation to Python.

$$f(x) \longrightarrow 3x + 2$$

# Translating an algebraic function to Python

First, we must know how to write Python code! The first thing we will learn is that our algebraic expressions can also be written in Python, meaning the same thing. Consider the following function, and an attempted translation to Python.

$$f(x) \longrightarrow 3x + 2$$

```python
def f(x):
  3*x + 2
```

# Translating an algebraic function to Python

$$f(x) \longrightarrow 3x + 2$$

```python
def f(x):
  3*x + 2
```

There are some comments to be made about this function definition in Python.

# Translating an algebraic function to Python

$$f(x) \longrightarrow 3x + 2$$

```python
def f(x):
  3*x + 2
```

There are some comments to be made about this function definition in Python.

The first thing we notice is the inclusion of the word `def` before the function name. `def` is a keyword used to specify that a function is being defined.

# Translating an algebraic function to Python

$f(x) \longrightarrow 3x + 2$

```
def f(x):
  3*x + 2
```

There are some comments to be made about this function definition in Python.

The first thing we notice is the inclusion of the word `def` before the function name. `def` is a keyword used to specify that a function is being defined.

The second thing we notice is that the body of the function is indented relative to the function name and parameter list.

# Translating an algebraic function to Python

$$f(x) \longrightarrow 3x + 2$$

```
def f(x):
  3*x + 2
```

There are some comments to be made about this function definition in Python.

The first thing we notice is the inclusion of the word `def` before the function name. `def` is a keyword used to specify that a function is being defined.

The second thing we notice is that the body of the function is indented relative to the function name and parameter list.

The third thing we notice is that if we try to execute this function it doesn't work!

# Components of a Python function

```
def g(x, y):
  z = x/2 + 3
  return z**y - x
```

# Components of a Python function

Keyword to begin function definition

```
def g(x, y):
  z = x/2 + 3
  return z**y - x
```

# Components of a Python function

Keyword to begin function definition

Function Name

```
def g(x, y):
    z = x/2 + 3
    return z**y - x
```

# Components of a Python function

Keyword to begin function definition

Parameter List

```
def g(x, y):
  z = x/2 + 3
  return z**y - x
```

Function Name

# Components of a Python function

Keyword to begin function definition

Parameter List

```python
def g(x, y):
    z = x/2 + 3
    return z**y - x
```

Function Name

Function Body

# Components of a Python function

Keyword to begin function definition

Parameter List

```
def g(x, y):
    z = x/2 + 3
    return z**y - x
```

Function Name

Function Body

Specifies value to produce

# Function definition syntax

We have now seen some of the syntax to define a function, but why are some of these elements here?

## Function definition syntax

We have now seen some of the syntax to define a function, but why are some of these elements here?

- The `def` keyword is necessary to allow the interpreter to know it should begin to read a function definition

# Function definition syntax

We have now seen some of the syntax to define a function, but why are some of these elements here?

- The `def` keyword is necessary to allow the interpreter to know it should begin to read a function definition
- The function body must be indented underneath the *function header* so that the interpreter knows when code is no longer part of this function (when outdented).

## Function definition syntax

We have now seen some of the syntax to define a function, but why are some of these elements here?

- The `def` keyword is necessary to allow the interpreter to know it should begin to read a function definition
- The function body must be indented underneath the *function header* so that the interpreter knows when code is no longer part of this function (when outdented).
- The `return` keyword is needed to specify which value the function should produce. Since Python functions are series of *statements* executed sequentially, we must specify which expression should be produced as our final value.

# What is a statement?

As mentioned on the previous slide, a Python function is a series of statements executed sequentially. This is actually true of an entire Python program itself!

# What is a statement?

As mentioned on the previous slide, a Python function is a series of statements executed sequentially. This is actually true of an entire Python program itself!

Python statements each must occur on their own line. i.e. there is only one statement per line.

# What is a statement?

As mentioned on the previous slide, a Python function is a series of statements executed sequentially. This is actually true of an entire Python program itself!

Python statements each must occur on their own line. i.e. there is only one statement per line.

So far we've only discussed expressions in Python, so what is a statement?

# What is a statement?

As mentioned on the previous slide, a Python function is a series of statements executed sequentially. This is actually true of an entire Python program itself!

Python statements each must occur on their own line. i.e. there is only one statement per line.

So far we've only discussed expressions in Python, so what is a statement?

Technically the Python grammar specifies seventeen kinds of simple statements, we will only learn a few for now.

# Expression statements

Our simplest type of statement in Python is an expression. In our first non-working Python function we had the line `3*x + 2`. While that function didn't do what we expected, we did not receive an error from Python. That is because an expression itself is a valid statement itself!

# Expression statements

Our simplest type of statement in Python is an expression. In our first non-working Python function we had the line `3*x + 2`. While that function didn't do what we expected, we did not receive an error from Python. That is because an expression itself is a valid statement itself!

When an expression statement is executed, the expression is evaluated and the result is simply thrown away.

# Expression statements

Our simplest type of statement in Python is an expression. In our first non-working Python function we had the line `3*x + 2`. While that function didn't do what we expected, we did not receive an error from Python. That is because an expression itself is a valid statement itself!

When an expression statement is executed, the expression is evaluated and the result is simply thrown away.

What could possibly be the use of evaluating an expression just to throw the result away? In an ideal world, nothing.

# Expression statements

Our simplest type of statement in Python is an expression. In our first non-working Python function we had the line `3*x + 2`. While that function didn't do what we expected, we did not receive an error from Python. That is because an expression itself is a valid statement itself!

When an expression statement is executed, the expression is evaluated and the result is simply thrown away.

What could possibly be the use of evaluating an expression just to throw the result away? In an ideal world, nothing. However, as Python is unfortunately not an ideal world we will see use cases of these later.

# Assignment statement

In mathematics, we often write *let expressions* that specify the value of a variable.

$$\text{let } x = 5$$
$$\text{let } y = 12$$
$$\text{in } x^y + 3x + 2y + 17$$

Which indicates in that expression that `x` has the value of 5 and `y` has the value of 12.

In Python, we will *assign* the value of a variable using the =, similar to how the = in a let expression constrains the value of that free variable.

# Assignment statement

We saw an assignment statement in our function g above.

```
z = x/2 + 3
```

# Assignment statement

We saw an assignment statement in our function g above.

Variable to be assigned

```
z = x/2 + 3
```

## Assignment statement

We saw an assignment statement in our function g above.

Variable to be assigned     Expression to assign to

```
z = x/2 + 3
```

# Assignment statement

We saw an assignment statement in our function g above.

Variable to be assigned    Expression to assign to

```
z = x/2 + 3
```

Assignment Token

# Assignment statement

We saw an assignment statement in our function g above.

$$\overbrace{\texttt{z}}^{\text{Variable to be assigned}} \texttt{ = } \overbrace{\texttt{x/2 + 3}}^{\text{Expression to assign to}}$$

This stores the result of $\frac{x}{2} + 3$ in the variable z.

# Return Statement

The `return` statement is used to specify the value that a function should produce as its result.

When a `return` statement executes, the currently executing function call immediately ends and produces its result to the calling expression.

Functions may have multiple `return` statements, but only the first one to execute in any given call will have an effect per above.

# Table of Contents

Writing Simple Functions in Python

Our first Python Function

Python Statements

Function Specification and Comments

Conditional Execution

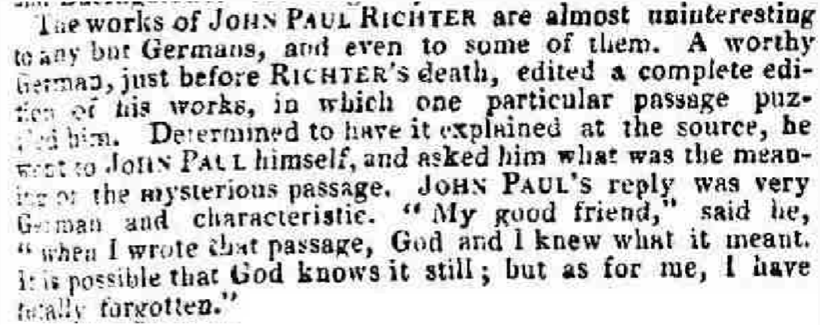
Boolean Logic

Understanding Python Conditionals

# The need for documentation

The works of JOHN PAUL RICHTER are almost uninteresting to any but Germans, and even to some of them. A worthy German, just before RICHTER's death, edited a complete edition of his works, in which one particular passage puzzled him. Determined to have it explained at the source, he went to JOHN PAUL himself, and asked him what was the meaning of the mysterious passage. JOHN PAUL's reply was very German and characteristic. "My good friend," said he, "when I wrote that passage, God and I knew what it meant. It is possible that God knows it still; but as for me, I have totally forgotten."

## The need for documentation


The works of JOHN PAUL RICHTER are almost uninteresting to any but Germans, and even to some of them. A worthy German, just before RICHTER's death, edited a complete edition of his works, in which one particular passage puzzled him. Determined to have it explained at the source, he went to JOHN PAUL himself, and asked him what was the meaning of the mysterious passage. JOHN PAUL's reply was very German and characteristic. "My good friend," said he, "when I wrote that passage, God and I knew what it meant. It is possible that God knows it still; but as for me, I have totally forgotten."

"My good friend, when I wrote that passage, God and I knew what it meant. It is possible that God knows it still; but as for me, I have totally forgotten." — Jean Paul Richter

From The Morning Chronicle, Issue 17755, August 9th, 1826, London England.

# The need for documentation

```
// Dear programmer:
// When I wrote this code, only god and
// I knew how it worked.
// Now, only god knows it!
```

# The need for documentation

```
// Dear programmer:
// When I wrote this code, only god and
// I knew how it worked.
// Now, only god knows it!
```

— An old programming joke, attribution unknown.

# What are comments?

Comments are simply text embedded into the source code of a program that is *not* part of the program and is meant to be ignored by the tool performing our translation into machine code.

## What are comments?

Comments are simply text embedded into the source code of a program that is *not* part of the program and is meant to be ignored by the tool performing our translation into machine code.

As comments are meant entirely to be ignored by the translator, their only purpose is to convey information to human readers of the source code.

# What are comments?

Comments are simply text embedded into the source code of a program that is *not* part of the program and is meant to be ignored by the tool performing our translation into machine code.

As comments are meant entirely to be ignored by the translator, their only purpose is to convey information to human readers of the source code.

Programmers have many opinions about how to use comments, and there are many *style guides* that suggest difference uses of them. We will follow one specific to this course.

## Comment syntax in Python

In Python we have several ways to write a comment.

```
# Octothorpes can be used to write a single-line comment
# We won't use these as much...

'''
Text enclosed between three single quotes can be used
to write a multi-line comment. These are also called
docstrings by the Python documentation.

We will use these heavily!
'''
```

# The CMPUT274 function specification

In this class *every* function you write will require a multi-line comment written at the beginning of it in a specific format. This is a part of our CMPUT274 style guide. This format will grow as new parts of it become necessary.

The purpose of this comment is to provide the *specification* of the function. This tells readers of your code exactly how the function is meant to be used, and expresses information like the domain and codomain.

# Example function specification

```
def pigLatinCons(s):
    '''
    pigLatinCons produces the pig latin version of
    a string that begins with a consonant.

    s - An alphabetic string that begins
        with a consonant
    Returns - An alphabetic string

    Examples:
    pigLatinCons("hello") -> "ellohay"
    pigLatinCons("brains") -> "rainsbay"
    '''
    return s[1:] + s[0] + "ay"
```

# CMPUT274 current function specification

Currently our function specification contains three components, in a specific order. They are

- A description of the functions purpose
- A list detailing the expected types of each parameter, and the resultant return type
- A list of example function applications

It may seem silly for small functions, but writing this specification before writing a function can help you to clarify exactly what purpose a function has and give some insights into its requirements.

# Table of Contents


Writing Simple Functions in Python

- Our first Python Function
- Python Statements

Function Specification and Comments

Conditional Execution

Boolean Logic

Understanding Python Conditionals

# The Pig Latin function

We previously defined the Pig Latin function *pl* as follows

$$pl(c_1...c_n) = \begin{cases} c_2...c_n c_1 ay & \text{if } c_1 \text{is a consonant} \\ c1_{.}..c_n way & \text{otherwise} \end{cases}$$

The Python function `pigLatinCons` only handles the first case.
How can we write a function like *pl* whose behaviour varies based on the argument provided?

# Execution control flow

We have said our functions execute each statement sequentially. However, sometimes we want to choose between multiple sets of statements to be executed — this is called conditional execution.

We achieve this in Python with the help of `if` statements.

## Simple `if` statement format

The format of our simplest type of `if` statement is as follows

```
if expr:
  stmt1
  ...
  stmtn
```

# Simple `if` statement format

The format of our simplest type of `if` statement is as follows

```
if expr:
  stmt1
  ...
  stmtn
```

Much like a function definition we use indentation to specify which code should be guarded by this `if` condition. We can have one or more statements guarded by an `if` condition.

# Simple `if` statement format

The format of our simplest type of `if` statement is as follows

```
if expr:
  stmt1
  ...
  stmtn
```

Much like a function definition we use indentation to specify which code should be guarded by this `if` condition. We can have one or more statements guarded by an `if` condition.

What is the purpose of the `expr` after the `if` keyword? Its evaluation determines whether the guarded code will be executed.

## Simple `if` statement format

The format of our simplest type of `if` statement is as follows

```
if expr:
  stmt1
  ...
  stmtn
```

Much like a function definition we use indentation to specify which code should be guarded by this `if` condition. We can have one or more statements guarded by an `if` condition.

What is the purpose of the `expr` after the `if` keyword? Its evaluation determines whether the guarded code will be executed. What values will lead to the guarded code being executed or not?

# The `bool` type

We have a type in Python for representing *truth values*. This type is called the `bool`[1]type.

[1]Short for *boolean*, named for the mathematician George Boole

## The `bool` type

We have a type in Python for representing *truth values*. This type is called the `bool`[1]type.

There are only two values that belong to the type `bool`. They are `True` and `False`. These also happen to be two keywords, they behave as literal values of the `bool` type.

[1]Short for *boolean*, named for the mathematician George Boole

# The `bool` type

We have a type in Python for representing *truth values*. This type is called the `bool`[1]type.

There are only two values that belong to the type `bool`. They are `True` and `False`. These also happen to be two keywords, they behave as literal values of the `bool` type.

When the guard expression of an `if` statement evaluates to `True` the guarded statements will be executed. When the guard expression evaluates to `False` the guarded statements will not be executed. What kind of expressions evaluate to `True` or `False`?

[1]Short for *boolean*, named for the mathematician George Boole

# Comparison operators

We have several operators in Python known as *comparison* operators. These operators are binary operators that compare their operands and evaluate to `True` or `False`.

| **Operator** | **Comparison performed** |
|---|---|
| `==` | Equality |
| `!=` | Inequality |
| `>` | Greater than |
| `<` | Less than |
| `>=` | Greater than or equal to |
| `<=` | Less than or equal to |

# Example comparison expressions

$$
\begin{aligned}
5 == 3 &\longrightarrow \textit{False} \\
25 >= 17 &\longrightarrow \textit{True} \\
22 < 22 &\longrightarrow \textit{False} \\
22 <= 22 &\longrightarrow \textit{True}
\end{aligned}
$$

# Comparisons on strings

In order to write the *pl* function we need conditional execution, but our condition is based on *text*. How can we state the question “is the first letter of this string a vowel?” in Python?

# Comparisons on strings

In order to write the *pl* function we need conditional execution, but our condition is based on *text*. How can we state the question "is the first letter of this string a vowel?" in Python?

Can we use our comparison operators on text?

# Comparison operators on strings

Our comparison operators *can* be used on sequences.

The equality and inequality operators behave as once would expect. Equality produces true if and only if two sequences contain the same items in the same order. Inequality produces true if and only if two sequences do not contain the same items in the same order.

The less-than and greater-than operators require a bit more explanation.

# Lexicographic ordering

Lexicographic ordering is a generalization of the concept used for alphabetical ordering. The properties are similar to that which one would understand for alphabetical ordering, but can be abstracted to sequences of any symbols which have an ordering.

That is, there exists a position in $x$ and $y$ where the item at that position in $x$ is ordered first, and all items before that position in $x$ were either the same as $y$ or also came before the item in $y$. The above can be read as Similar definitions apply to $\leq$, $>$, $\geq$.

# Lexicographic ordering

Lexicographic ordering is a generalization of the concept used for alphabetical ordering. The properties are similar to that which one would understand for alphabetical ordering, but can be abstracted to sequences of any symbols which have an ordering.

Let $x$ and $y$ be the sequences $a_1 \ldots a_n$ and $b_1 \ldots b_n$ respectively and the individual items of these collections be *ordered symbols* of the same set. Then we have the following property That is, there exists a position in $x$ and $y$ where the item at that position in $x$ is ordered first, and all items before that position in $x$ were either the same as $y$ or also came before the item in $y$. The above can be read as Similar definitions apply to $\leq$, $>$, $\geq$.

# Lexicographic ordering

Lexicographic ordering is a generalization of the concept used for alphabetical ordering. The properties are similar to that which one would understand for alphabetical ordering, but can be abstracted to sequences of any symbols which have an ordering.

Let $x$ and $y$ be the sequences $a_1...a_n$ and $b_1...b_n$ respectively and the individual items of these collections be *ordered symbols* of the same set. Then we have the following property

$$\exists i \in \mathbb{N}(a_i < b_i \text{ and } \forall j < i, a_j \leq b_j) \iff x < y$$

That is, there exists a position in $x$ and $y$ where the item at that position in $x$ is ordered first, and all items before that position in $x$ were either the same as $y$ or also came before the item in $y$. The above can be read as Similar definitions apply to $\leq$, $>$, $\geq$.

# Examples of lexicographic ordering

Consider the two sequences $x = (1, 2, 8, 12)$ and $y = (1, 2, 9, 2)$. It is the case that $x < y$.

The third item of $x$ is less than the third item of $y$, and all items before the third item are equal.

More simply put for two sequences $x$ and $y$, $x < y$ if the first item in order that differs between $x$ and $y$ is less in $x$ than in $y$.

# Writing the pl function — first solution

Given what we have learned so far we can now write one implementation of our *pl* function.

Since we already have the `pigLatinCons` function we will also write a `pigLatinVowel` function to aid us in our solution.

Functions that we build specifically to help with the implementation of another larger function are called *helper functions*. We can say that `pigLatinCons` and `pigLatinVowel` are helper functions for the function *pl*.

```
def pigLatinVowel(s):
  '''
  pigLatinVowel produces the pig latin version of
  a string that begins with a vowel.

  s - An alphabetic string that begins
      with a vowel
  Returns - An alphabetic string

  Examples:
  pigLatinVowel("orange") -> "orangeway"
  pigLatinVowel("irate") -> "irateway"
  '''
  return s + "way"
```

# First `pl` implementation

```
def pl(s):
  if s[0] == "a":
    return pigLatinVowel(s)
  if s[0] == "e":
    return pigLatinVowel(s)
  if s[0] == "i":
    return pigLatinVowel(s)
  if s[0] == "o":
    return pigLatinVowel(s)
  if s[0] == "u":
    return pigLatinVowel(s)
  return pigLatinCons(s)
```

# How does `pl` work?

Our first `pl` function combines several things we've learned. Some of these are:

- Indexing can be used to extract one letter of a string

# How does `p1` work?

Our first `p1` function combines several things we've learned. Some of these are:

- Indexing can be used to extract one letter of a string
- Strings can be compared directly

## How does `pl` work?

Our first `pl` function combines several things we've learned. Some of these are:

- Indexing can be used to extract one letter of a string
- Strings can be compared directly
- Python functions are executed sequentially, and whenever any `return` statement is executed the function application immediately finishes producing the result

# How does `pl` work?

Our first `pl` function combines several things we've learned. Some of these are:

- Indexing can be used to extract one letter of a string
- Strings can be compared directly
- Python functions are executed sequentially, and whenever any `return` statement is executed the function application immediately finishes producing the result
- An `if` condition can be used to conditionally execute code

## How does `pl` work?

Our first `pl` function combines several things we've learned. Some of these are:

- Indexing can be used to extract one letter of a string
- Strings can be compared directly
- Python functions are executed sequentially, and whenever any `return` statement is executed the function application immediately finishes producing the result
- An `if` condition can be used to conditionally execute code

However, our function currently has some problems.

## Problem with `pl`

The problem our `pl` function has is that it does not work properly for capital letters.

# Problem with `pl`

The problem our `pl` function has is that it does not work properly for capital letters.

If a string is entered that begins with a capital vowel then we will produce the wrong value — returning the result of `pigLatinCons` when we should produce the result of `pigLatinVowel`.

## Problem with `pl`

The problem our `pl` function has is that it does not work properly for capital letters.

If a string is entered that begins with a capital vowel then we will produce the wrong value — returning the result of `pigLatinCons` when we should produce the result of `pigLatinVowel`.

We could solve this problem simply by adding several more `if` conditions, one for each uppercase vowel.

## Problem with `pl`

The problem our `pl` function has is that it does not work properly for capital letters.

If a string is entered that begins with a capital vowel then we will produce the wrong value — returning the result of `pigLatinCons` when we should produce the result of `pigLatinVowel`.

We could solve this problem simply by adding several more `if` conditions, one for each uppercase vowel.

The other problem with our function is that it is absolutely horrendous from a design perspective. This we will talk about and solve later!

# Practice Problem

Write the function `incomeTax` that has one parameter that represents a Canadian citizen's taxable income and returns the amount of federal income tax that person owes. Federal income tax is determined based on the following rates

- 14.5% is taxed on the first 57375 CAD earned
- 20.5% is taxed on the next 57375 CAD earned above the former
- 26% is taxed on the next 63132 CAD earned above the former
- 29% is taxed on the next 75532 CAD earned above the former
- 33% is taxed on all income over that

Challenge: Write the function in such a way that the `incomeTax` function itself has *no* conditional execution in its own code. Hint — helper functions!

# Table of Contents


Writing Simple Functions in Python

- Our first Python Function
- Python Statements

Function Specification and Comments

Conditional Execution

Boolean Logic

Understanding Python Conditionals

# Boolean operators

In addition to comparison operators producing `bool` values, we also have three *boolean operators* which operator directly on boolean expressions.

# Boolean operators

In addition to comparison operators producing `bool` values, we also have three *boolean operators* which operator directly on boolean expressions.

These operators are `and`, `or`, & `not`. Each of these operators is a keyword that behaves as an operator. The first two are binary operators while the last is a unary operator.

# `and` operator

Logical *conjunction* (the logical `and` operator) produces `True` only when both of its operands are `True`, `False` otherwise.

# `and` operator

Logical *conjunction* (the logical `and` operator) produces `True` only when both of its operands are `True`, `False` otherwise.

Since there are only two possible truth values it is simple to display the behaviour of logical conjunction in a table.

# `and` **operator**

Logical *conjunction* (the logical `and` operator) produces `True` only when both of its operands are `True`, `False` otherwise.

Since there are only two possible truth values it is simple to display the behaviour of logical conjunction in a table.

The table below depicts the behaviour of the `and` operator in the expression `X and Y`

| `X` | `Y` | `X and Y` |
|---|---|---|
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

## `or` operator

Logical *disjunction* (the logical `or` operator) produces `True` when *at least one* of its operands is `True`, `False` otherwise.

## `or` operator

Logical *disjunction* (the logical `or` operator) produces `True` when *at least one* of its operands is `True`, `False` otherwise.

Logical *disjunction* (the logical `or` operator) produces `True` when *at least one* of its operands is `True`, `False` otherwise.

The table below depicts the behaviour of the `or` operator in the expression `X or Y`

| `X` | `Y` | `X or Y` |
|---|---|---|
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

## `not` **operator**

Logical *negation* (the logical `not` operator) produces `True` when its operand is `False`, `False` otherwise.

## `not` operator

Logical *negation* (the logical `not` operator) produces `True` when its operand is `False`, `False` otherwise.

## `not` operator

Logical *negation* (the logical `not` operator) produces `True` when its operand is `False`, `False` otherwise.

The table below depicts the behaviour of the `not` operator in the expression `not X`

| `X` | `not X` |
|---|---|
| True | False |
| True | True |

# Complex logical expressions

It is important to note that logical expressions are still just that — *expressions*.

# Complex logical expressions

It is important to note that logical expressions are still just that — *expressions*.

As such, the operands of each logical expression are themselves expressions. With that rises more complex logical expressions, for example:

# Complex logical expressions

It is important to note that logical expressions are still just that — *expressions*.

As such, the operands of each logical expression are themselves expressions. With that rises more complex logical expressions, for example:

`not (x or (y and z)) and (z and not (not x or y))`

# Complex logical expressions

It is important to note that logical expressions are still just that — *expressions*.

As such, the operands of each logical expression are themselves expressions. With that rises more complex logical expressions, for example:

```
not (x or (y and z)) and (z and not (not x or y))
```

Can you tell what this expression evaluates to? You may think you need to know the values of `x`, `y`, and `z` — you don't though!

## Boolean algebra

Boolean algebra is an algebra that operates only over the truth values *true* and *false*. We will often represent these values as 1 for true and 0 for false.

Additionally, we have symbols for representing conjunction, disjunction, and negation.

| **English Name** | **English Operator** | **Symbol** |
|---|---|---|
| Disjunction | or | $\vee$ |
| Conjunction | and | $\wedge$ |
| Negation | not | $\neg$ |

# Boolean algebra

Boolean algebra is an algebra that operates only over the truth values *true* and *false*. We will often represent these values as `1` for true and `0` for false.

Additionally, we have symbols for representing conjunction, disjunction, and negation.

| **English Name** | **English Operator** | **Symbol** |
|---|---|---|
| Disjunction | or | $\vee$ |
| Conjunction | and | $\wedge$ |
| Negation | not | $\neg$ |

There are many different methods for representing boolean algebra operations, the one shown above is one choice. It is the one we will use in this class.

## Boolean algebra laws

Just as there are algebraic laws of arithmetic, so too are there boolean algebraic laws.

Learning these laws can help us to take complex boolean expressions and simplify them. The boolean algebra laws are fairly intuitive, as they follow logic we often see in our day-to-day life.

# Identity and annihilator laws

Due to the nature of $\wedge$ and $\vee$ some properties are immediately obvious when constant values are included.

$$
\begin{aligned}
x \vee 0 &= x && \textit{Identity} \\
x \wedge 1 &= x && \textit{Identity} \\
x \vee 1 &= 1 && \textit{Annihilator} \\
x \wedge 0 &= 0 && \textit{Annihilator}
\end{aligned}
$$

# Associativity

Both $\wedge$ and $\vee$ are associative. That is:

$$x \vee (y \vee z) = (x \vee y) \vee z$$
$$x \wedge (y \wedge z) = (x \wedge y) \wedge z$$

So between strict disjunctions or conjunctions we can drop any parentheses.

# Commutativity

Both $\wedge$ and $\vee$ are commutative. That is:

$$x \vee y = y \vee x$$
$$x \wedge y = y \wedge x$$

This means we can swap the operands of a conjunction or disjunction if it helps us in simplifying our expression.

# Distributivity

Just as in arithmetic algebra we can distribute operators over parentheticals.

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$
$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$$

# Idempotence

An idempotent operation is one that has the same result whether it is applied once or many times. Multiplying by 1 or 0 is an example of an idempotent operation, as is adding 0.

$$x \vee x = x$$

$$x \wedge x = x$$

This law is helpful in eliminating redundant terms.

# Complement laws

The inclusion of our negation operator adds some important laws for simplification.

$$x \wedge \neg x = 0$$
$$x \vee \neg x = 1$$

These laws are extremely important for simplification, as they allow us to introduce constants into an expression of only variables under the right circuimstances.

# Double negation

You have probably heard a double negation used in natural language before. Just as it does in natural language a double negative cancels itself out in boolean algebra.

$$\neg\neg x = x$$

# De Morgan's laws

De Morgan's laws are laws for distributing a negation over a conjunction or disjunction.

$$\neg(x \wedge y) = \neg x \vee \neg y$$
$$\neg(x \vee y) = \neg x \wedge \neg y$$

# Example simplification

Let's now simplify our expression from earlier.

$$\neg(x \vee (y \wedge z)) \wedge (z \wedge \neg(\neg x \vee y))$$

# Example simplification

Let's now simplify our expression from earlier.

$$\neg(x \vee (y \wedge z)) \wedge (z \wedge \neg(\neg x \vee y))$$

$$\neg x \wedge \neg(y \wedge z) \wedge (z \wedge \neg(\neg x \vee y)) \qquad \textit{De Morgan's}$$

# Example simplification

Let's now simplify our expression from earlier.

$$
\begin{aligned}
&\neg(x \lor (y \land z)) \land (z \land \neg(\neg x \lor y)) && \\
&\neg x \land \neg(y \land z) \land (z \land \neg(\neg x \lor y)) && \textit{De Morgan's} \\
&\neg x \land \neg(y \land z) \land z \land \neg(\neg x \lor y) && \textit{Associativity}
\end{aligned}
$$

# Example simplification

Let's now simplify our expression from earlier.

$$
\begin{aligned}
&\neg(x \vee (y \wedge z)) \wedge (z \wedge \neg(\neg x \vee y)) && \\
&\neg x \wedge \neg(y \wedge z) \wedge (z \wedge \neg(\neg x \vee y)) && \textit{De Morgan's} \\
&\neg x \wedge \neg(y \wedge z) \wedge z \wedge \neg(\neg x \vee y) && \textit{Associativity} \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge (\neg\neg x \wedge \neg y) && \textit{De Morgan's} \times 2
\end{aligned}
$$

# Example simplification

Let's now simplify our expression from earlier.

$$
\begin{aligned}
&\neg(x \vee (y \wedge z)) \wedge (z \wedge \neg(\neg x \vee y)) && \\
&\neg x \wedge \neg(y \wedge z) \wedge (z \wedge \neg(\neg x \vee y)) && \textit{De Morgan's} \\
&\neg x \wedge \neg(y \wedge z) \wedge z \wedge \neg(\neg x \vee y) && \textit{Associativity} \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge (\neg\neg x \wedge \neg y) && \textit{De Morgan's} \times 2 \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge (x \wedge \neg y) && \textit{Double Negation}
\end{aligned}
$$

## Example simplification

Let's now simplify our expression from earlier.

$$
\begin{aligned}
&\neg(x \vee (y \wedge z)) \wedge (z \wedge \neg(\neg x \vee y)) && \\
&\neg x \wedge \neg(y \wedge z) \wedge (z \wedge \neg(\neg x \vee y)) && \textit{De Morgan's} \\
&\neg x \wedge \neg(y \wedge z) \wedge z \wedge \neg(\neg x \vee y) && \textit{Associativity} \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge (\neg\neg x \wedge \neg y) && \textit{De Morgan's} \times 2 \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge (x \wedge \neg y) && \textit{Double Negation} \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge x \wedge \neg y && \textit{Associativity}
\end{aligned}
$$

# Example simplification

Let's now simplify our expression from earlier.

$$
\begin{aligned}
&\neg(x \vee (y \wedge z)) \wedge (z \wedge \neg(\neg x \vee y)) & \\
&\neg x \wedge \neg(y \wedge z) \wedge (z \wedge \neg(\neg x \vee y)) & \textit{De Morgan's} \\
&\neg x \wedge \neg(y \wedge z) \wedge z \wedge \neg(\neg x \vee y) & \textit{Associativity} \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge (\neg\neg x \wedge \neg y) & \textit{De Morgan's} \times 2 \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge (x \wedge \neg y) & \textit{Double Negation} \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge x \wedge \neg y & \textit{Associativity} \\
&\neg x \wedge x \wedge (\neg y \vee \neg z) \wedge z \wedge \neg y & \textit{Commutativity}
\end{aligned}
$$

## Example simplification

Let's now simplify our expression from earlier.

$$
\begin{aligned}
&\neg(x \vee (y \wedge z)) \wedge (z \wedge \neg(\neg x \vee y)) && \\
&\neg x \wedge \neg(y \wedge z) \wedge (z \wedge \neg(\neg x \vee y)) && \textit{De Morgan's} \\
&\neg x \wedge \neg(y \wedge z) \wedge z \wedge \neg(\neg x \vee y) && \textit{Associativity} \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge (\neg\neg x \wedge \neg y) && \textit{De Morgan's} \times 2 \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge (x \wedge \neg y) && \textit{Double Negation} \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge x \wedge \neg y && \textit{Associativity} \\
&\neg x \wedge x \wedge (\neg y \vee \neg z) \wedge z \wedge \neg y && \textit{Commutativity} \\
&0 \wedge (\neg y \vee \neg z) \wedge z \wedge \neg y && \textit{Annihilator}
\end{aligned}
$$

# Example simplification

Let's now simplify our expression from earlier.

$$
\begin{aligned}
&\neg(x \vee (y \wedge z)) \wedge (z \wedge \neg(\neg x \vee y)) && \\
&\neg x \wedge \neg(y \wedge z) \wedge (z \wedge \neg(\neg x \vee y)) && \textit{De Morgan's} \\
&\neg x \wedge \neg(y \wedge z) \wedge z \wedge \neg(\neg x \vee y) && \textit{Associativity} \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge (\neg\neg x \wedge \neg y) && \textit{De Morgan's} \times 2 \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge (x \wedge \neg y) && \textit{Double Negation} \\
&\neg x \wedge (\neg y \vee \neg z) \wedge z \wedge x \wedge \neg y && \textit{Associativity} \\
&\neg x \wedge x \wedge (\neg y \vee \neg z) \wedge z \wedge \neg y && \textit{Commutativity} \\
&0 \wedge (\neg y \vee \neg z) \wedge z \wedge \neg y && \textit{Annihilator} \\
&0 && \textit{Annihilator} \times 3
\end{aligned}
$$

# Boolean logic exercise

Knowledge Check: Can you simplify the following boolean expressions? Even if you can't come to a final truth value try to eliminate as many terms as you can.

$$\neg(x \wedge y) \vee y$$
$$x \wedge (y \vee (\neg x \vee z))$$

# Boolean logic exercise

Knowledge Check: Can you simplify the following boolean expressions? Even if you can't come to a final truth value try to eliminate as many terms as you can.

$$\neg(x \wedge y) \vee y = 1$$

$$x \wedge (y \vee (\neg x \vee z)) = x \wedge (y \vee z)$$

# Normal forms

What does it mean to “simplify” a boolean expression that can't be outright solved?

At a conceptual level it means we want it in a form that's easier to parse or solve later. Forms that are simple are ones where the terms are easy to move around based on our rules.

If we have a series of disjunctions of non-disjunctive expressions, or a series of conjunctions of non-conjunctive expressions then we have a fairly easy to work with expression. These two possibilities describe our two normal forms.

# Disjunctive normal form

A boolean expression is said to be in *disjunctive normal form* (DNF) if it comprises only disjunctions of sub-expressions, where the sub-expressions can only contain conjunctions.

$$(\neg x \land y \land z) \lor (p \land q) \lor t \qquad \textit{An expression in DNF}$$

$$(\neg x \land y \lor z) \lor (p \land q) \lor t \qquad \textit{An expression not in DNF}$$

# Boolean algebra precedence

There is no agreed upon precedence of boolean operators — that means an expression such as $x \wedge y \vee z$ is ambiguous without further elaboration[2].

So we define our precedence in this class from highest to lowest as $()$, $\neg$, $\wedge$, $\vee$.

[2] Note that this is not really a failure of the algebra, but of the individual presenting an expression. This representation would need either rules specified, or more parentheses. Other representations encode the precedence implicitly.

# Table of Contents


Writing Simple Functions in Python

- Our first Python Function
- Python Statements

Function Specification and Comments

Conditional Execution

Boolean Logic

Understanding Python Conditionals

## Python conditional forms

So far we have only seen the form of a conditional that uses an `if`

```
if expr:
  stmt1
  ...
  stmtn
```

However, we have a few other language constructs for conditional execution.

# `elif` clause

Python has an `elif` clause which must follow either an `if` or an `elif`.

```
if expr1:
  stmts1
elif expr2:
  stmts2
elif expr3:
  stmts3
...
elif exprN:
  stmtsN
```

## `if/elif` chain

When there are `elif` statements following an `if` statement their conditions are checked in the order in which they appear.

The first condition whose expression evaluates to `True` in order has its statements evaluated, all other statements are skipped regardless of if the guarding expression was true or not.

# `if` `elif` **example**

Consider the following function definition, and evaluate each of the given function applications.

# if elif example

Consider the following function definition, and evaluate each of the given function applications.

```
def foo(a, b, c, x):
  if a:
    z = x*5
  elif b:
    z = x-20
  elif c:
    z = x+1
  return z
```

# `if elif` example

Consider the following function definition, and evaluate each of the given function applications.

```
def foo(a, b, c, x):
  if a:
    z = x*5
  elif b:
    z = x-20
  elif c:
    z = x+1
  return z
```

- `foo(True, True, False, 5)`
- `foo(False, True, True, 7)`
- `foo(True, True, True, 10)`

# `if elif` example

Consider the following function definition, and evaluate each of the given function applications.

```
def foo(a, b, c, x):
  if a:
    z = x*5
  elif b:
    z = x-20
  elif c:
    z = x+1
  return z
```

- `foo(True, True, False, 5)` $\longrightarrow$ 25
- `foo(False, True, True, 7)`
- `foo(True, True, True, 10)`

# `if elif` example

Consider the following function definition, and evaluate each of the given function applications.

```
def foo(a, b, c, x):
  if a:
    z = x*5
  elif b:
    z = x-20
  elif c:
    z = x+1
  return z
```

- `foo(True, True, False, 5)` $\longrightarrow 25$
- `foo(False, True, True, 7)` $\longrightarrow -13$
- `foo(True, True, True, 10)`

# `if elif` example

Consider the following function definition, and evaluate each of the given function applications.

```
def foo(a, b, c, x):
  if a:
    z = x*5
  elif b:
    z = x-20
  elif c:
    z = x+1
  return z
```

- `foo(True, True, False, 5)` $\longrightarrow 25$
- `foo(False, True, True, 7)` $\longrightarrow -13$
- `foo(True, True, True, 10)` $\longrightarrow 50$

## `if elif` example

Compare the resultant values of the previous function to this one to illustrate the significance of `elif`

# `if elif` example

Compare the resultant values of the previous function to this one to illustrate the significance of `elif`

```
def foo(a, b, c, x):
  if a:
    z = x*5
  if b:
    z = x-20
  if c:
    z = x+1
  return z
```

# `if elif` example

Compare the resultant values of the previous function to this one to illustrate the significance of `elif`

```
def foo(a, b, c, x):
  if a:
    z = x*5
  if b:
    z = x-20
  if c:
    z = x+1
  return z
```

- `foo(True, True, False, 5)`
- `foo(False, True, True, 7)`
- `foo(True, True, True, 10)`

# `if elif` example

Compare the resultant values of the previous function to this one to illustrate the significance of `elif`

```
def foo(a, b, c, x):
  if a:
    z = x*5
  if b:
    z = x-20
  if c:
    z = x+1
  return z
```

- `foo(True, True, False, 5)` $\longrightarrow -15$
- `foo(False, True, True, 7)`
- `foo(True, True, True, 10)`

# `if elif` example

Compare the resultant values of the previous function to this one to illustrate the significance of `elif`

```
def foo(a, b, c, x):
  if a:
    z = x*5
  if b:
    z = x-20
  if c:
    z = x+1
  return z
```

- `foo(True, True, False, 5)` $\longrightarrow -15$
- `foo(False, True, True, 7)` $\longrightarrow 8$
- `foo(True, True, True, 10)`

# `if elif` example

Compare the resultant values of the previous function to this one to illustrate the significance of `elif`

```
def foo(a, b, c, x):
  if a:
    z = x*5
  if b:
    z = x-20
  if c:
    z = x+1
  return z
```

- `foo(True, True, False, 5)` $\longrightarrow -15$
- `foo(False, True, True, 7)` $\longrightarrow 8$
- `foo(True, True, True, 10)` $\longrightarrow 11$

## Simulating `elif` with `if`

As only the *first* condition that is `True` in an `if/elif` chain has its statements executed, then it means an `elif` only has its code executed if all the preceding conditions were `False`. This means we can simulate the behaviour of `elif` with logical operations

## Simulating `elif` with `if`

As only the *first* condition that is `True` in an `if/elif` chain has its statements executed, then it means an `elif` only has its code executed if all the preceding conditions were `False`. This means we can simulate the behaviour of `elif` with logical operations

```
def foo(a, b, c, x):
  if a:
    z = x*5
  elif b:
    z = x-20
  elif c:
    z = x+1
  return z
```

# Simulating `elif` with `if`

As only the *first* condition that is `True` in an `if/elif` chain has its statements executed, then it means an `elif` only has its code executed if all the preceding conditions were `False`. This means we can simulate the behaviour of `elif` with logical operations

```
def foo(a, b, c, x):
  if a:
    z = x*5
  elif b:
    z = x-20
  elif c:
    z = x+1
  return z
```

```
def foo(a, b, c, x):
  if a:
    z = x*5
  if not a and b:
    z = x-20
  if not a and not b and c:
    z = x+1
  return z
```

# The `else` clause

The `else` clause must follow either an `if` or an `elif`.

## The `else` clause

The `else` clause must follow either an `if` or an `elif`.

```
if expr1:
  stmts1
elif expr2:
  stmts2
...
elif exprN:
  stmtsN
else:
  elseStmts
```

# The `else` clause

The `else` clause must follow either an `if` or an `elif`.

```
if expr1:
  stmts1
elif expr2:
  stmts2
...
elif exprN:
  stmtsN
else:
  elseStmts
```

```
if expr1:
  stmts1
else:
  elseStmts
```

# `else` clause behaviour

The `else` clause has its statements executed only if all prior conditions were `False`. If the `if` condition or any of the `elif` conditions were taken then the `else` will not be taken. In this way, `else` can also be simulated with boolean expressions.

# `else` clause behaviour

The `else` clause has its statements executed only if all prior conditions were `False`. If the `if` condition or any of the `elif` conditions were taken then the `else` will not be taken. In this way, `else` can also be simulated with boolean expressions.

```
if A:
  stmtsA
elif B:
  stmtsB
elif C:
  stmtsC
else:
  elseStmts
```

# `else` clause behaviour

The `else` clause has its statements executed only if all prior conditions were `False`. If the `if` condition or any of the `elif` conditions were taken then the `else` will not be taken. In this way, `else` can also be simulated with boolean expressions.

```
if A:
  stmtsA
elif B:
  stmtsB
elif C:
  stmtsC
else:
  elseStmts
```

```
if A:
  stmtsA
if not A and B:
  stmtsB
if not A and not B and C:
  stmtsC
if not A and not B and not C:
  elseStmts
```

## Practice Question

Write the function `milkPurchase` that returns one of the strings ``` ``Almond'' ```, ``` ``Oat'' ```, ``` ``Soy'' ```, or ``` ``Nothing'' ``` based on the following specification.

- If Almond milk is cheaper than 0.45 per 100ml then purchase Almond milk.
- If Oat milk is cheaper than 0.40 per 100ml then purchase Oat milk.
- If Soy milk is cheaper than 0.37 per 100ml then purchase Soy milk.
- Only one type of milk should be purchased.
- In the case that multiple milk conditions is met, the preference is to purchase Almond milk over Oat milk, and Oat milk over Soy milk.
- If any milk is less than 0.20 per 100ml then that milk has preference over any milk that costs more than 0.20 per 100ml.

# Nested conditions

Conditional statements can appear as part of the statements guarded by a conditional statement. We call these *nested conditionals*. An example function using nested conditionals is shown below.

```
def bikeToWork(temp, precip):
  if temp > 7:
    if precip > 0.4:
      return False
    else:
      return True
  else:
    return False
```

# Behaviour of nested conditionals

You know when a statement contained within a conditional is executed, and you know how a conditional statement is executed. There is nothing new to learn about how a nested conditional is executed, they behave according to the rules we've learned.

However, we should be careful to note that in the case of `elif` and `else` clauses they are tied to the immediately preceding conditional *at the same indentation level.*

# Confusing clauses

Consider the following function:

```
def playGame(hour, day):
  if day == "Monday" or day == "Wednesday":
    if hour < 23
      return True
  else:
    return False
```

What does this function return when the arguments 23 and ``Monday'' are provided?

## Confusing clauses

Consider the following function:

```
def playGame(hour, day):
  if day == "Monday" or day == "Wednesday":
    if hour < 23
      return True
  else:
    return False
```

What does this function return when the arguments 23 and ``Monday'' are provided?

Nothing! Something is wrong!

# Confusing clauses

```
def playGame(hour, day):
  if day == "Monday" or day == "Wednesday":
    if hour < 23
      return True
  else:
    return False
```

The `else` clause here is tied to the condition `if day == "Monday" or day == "Wednesday"`, because that condition is `True` the else clause never executes. However, the condition `if hour < 23` is not `True`, so the return statement underneath does not execute. This means this function never executes a `return` statement in this case, so it behaves like the very first function we tried to write!

# Simulating nested conditionals

Nested conditionals can also be simulated with boolean operations. Rewrite the `bikeToWork` function such that it has no nested conditionals.

# Simulating nested conditionals

Nested conditionals can also be simulated with boolean operations. Rewrite the `bikeToWork` function such that it has no nested conditionals.

```
def bikeToWork(temp, precip):
  if temp > 7:
    if precip > 0.4:
      return False
    else:
      return True
  else:
    return False
```

## Simulating nested conditionals

Nested conditionals can also be simulated with boolean operations. Rewrite the `bikeToWork` function such that it has no nested conditionals.

```
def bikeToWork(temp, precip):
  if temp > 7 and not precip > 0.4:
    return True
  elif temp > 7 and precip > 0.4:
    return False
  else:
    return False
```

# Simulating nested conditionals

Nested conditionals can also be simulated with boolean operations. Rewrite the `bikeToWork` function such that it has no nested conditionals.

```python
def bikeToWork(temp, precip):
  if temp > 7 and not precip > 0.4:
    return True
  else:
    return False
```

## The `leetSpeak` function

Now that we've got some functions under our belt, we start to consider more complex functions to write. One such function we've decided to implement is the `leetSpeak` function.

This function takes an alphabetical string and returns the result of translating that string to "leet speak".

# The rules of leet speak

The rules of leet speak are simple fairly simple, we will use the following ruleset

- The letter “E” is replaced with 3
- The letter “A” is replaced with 4
- The letter “L” is replaced with 1
- The letter “T” is replaced with 7
- The letter “S” is replaced with 5
- The letter “O” is replaced with 0
- If a word ends in the letter “ER” then the “ER” is dropped and “z0r” is appended to the end

## Leet speak examples

- leetspeak
- hacker
- awplord

# Leet speak examples

- leetspeak $\longrightarrow$ 13375p34k
- hacker
- awplord

# Leet speak examples

- leetspeak $\longrightarrow$ 13375p34k
- hacker $\longrightarrow$ h4ckz0r
- awplord

## Leet speak examples

- leetspeak $\longrightarrow$ 13375p34k
- hacker $\longrightarrow$ h4ckz0r
- awplord $\longrightarrow$ 4wpl0rd

## A problem — unknown string length

In order to implement the `leetSpeak` function we will need to compare every single character in the given string to our list of replaceable letters, we will also have to check the end of the string to determine if we should replace "ER" with "z0r".

# A problem — unknown string length

In order to implement the `leetSpeak` function we will need to compare every single character in the given string to our list of replaceable letters, we will also have to check the end of the string to determine if we should replace "ER" with "z0r".

We could write several conditions to check `s[0]`, `s[1]`, etc. How many of these do we write though?

## A problem — unknown string length

In order to implement the `leetSpeak` function we will need to compare every single character in the given string to our list of replaceable letters, we will also have to check the end of the string to determine if we should replace "ER" with "z0r".

We could write several conditions to check `s[0]`, `s[1]`, etc. How many of these do we write though?

The answer: we don't know! It depends on the length of the provided argument. This isn't something we can determine when writing the function. We need to be able to write some form of code that *repeats*.