Basics of using Python

Rob Hackman

Fall 2025

University of Alberta

Table of Contents

Makeup of a program

We have already said a program is a series of instructions for a computer to execute. When those instructions are written in a language that will ultimately be translated to machine code to execute we call that document *source code*.

Makeup of a program

We have already said a program is a series of instructions for a computer to execute. When those instructions are written in a language that will ultimately be translated to machine code to execute we call that document *source code*.

Source code is a series of symbols, which we typically call *tokens*, that has meaning in the given programming language. The order of the tokens must follow the rules of the language called the *syntax* of the language.

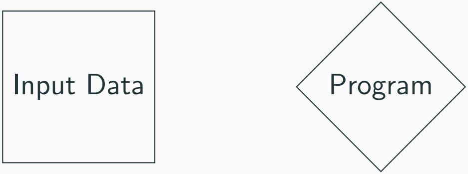
Purpose of a program

Every meaningful program takes input data, transforms it in some way, and produces output data.

Input Data

Purpose of a program

Every meaningful program takes input data, transforms it in some way, and produces output data.



Purpose of a program

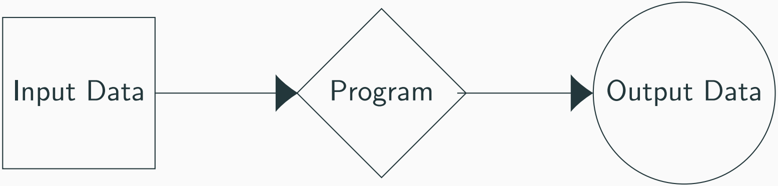
Every meaningful program takes input data, transforms it in some way, and produces output data.

Purpose of a program

Every meaningful program takes input data, transforms it in some way, and produces output data.

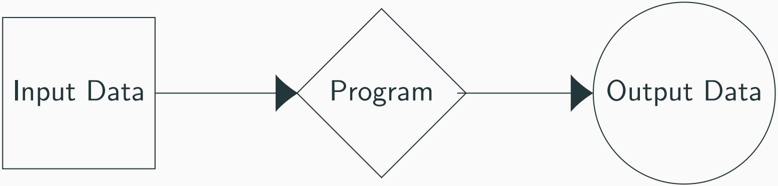
Purpose of a program

Every meaningful program takes input data, transforms it in some way, and produces output data.



Purpose of a program

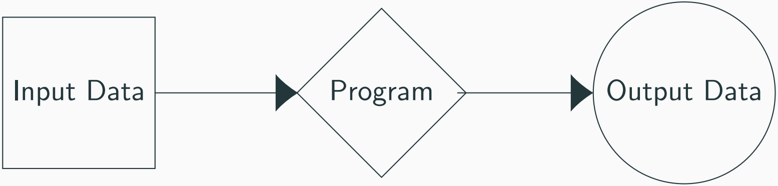
Every meaningful program takes input data, transforms it in some way, and produces output data.



The input and output data can be *anything*, not just numbers or text. For example input could be point clouds read from lidars and output could be actuation of a steering wheel and gas/brake pedals.

Purpose of a program

Every meaningful program takes input data, transforms it in some way, and produces output data.



The input and output data can be *anything*, not just numbers or text. For example input could be point clouds read from lidars and output could be actuation of a steering wheel and gas/brake pedals.

So a program takes input data that can be pretty much anything, and translates it into output data that could be pretty much anything. Seems familiar!

Table of Contents

Expressions

Expressions are an important part of any programming language.

Expressions

Expressions are an important part of any programming language.

Much like arithmetic expressions, a Python expression comprises numbers, variables, and operations.

Expressions

Expressions are an important part of any programming language.

Much like arithmetic expressions, a Python expression comprises numbers, variables, and operations.

Our expressions in Python can always be evaluated to a final value.

Atomic expressions

Our smallest type of expression is called an *atomic expression*. An atomic expression is one of the following types of expressions

Atomic expressions

Our smallest type of expression is called an *atomic expression*. An atomic expression is one of the following types of expressions

- A *literal*

Atomic expressions

Our smallest type of expression is called an *atomic expression*. An atomic expression is one of the following types of expressions

- A *literal*
- An *identifier*

Literal values

Tokens in a program have an abstract meaning prescribed to them by the rules of the language.

Literal values

Tokens in a program have an abstract meaning prescribed to them by the rules of the language.

While literal values, or just literals, also are part of the language and as such have their meaning prescribed by the rules of the language they also are meant to represent *literally* the value they appear as in our source code.

Literal values

Tokens in a program have an abstract meaning prescribed to them by the rules of the language.

While literal values, or just literals, also are part of the language and as such have their meaning prescribed by the rules of the language they also are meant to represent *literally* the value they appear as in our source code.

An example of a literal value in a Python program would be 25, which would represent the decimal number twenty-five.

Types of data

In programming our data comes in different *types*. The type of a particular piece of data is important because it dictates what sort of operations can be applied to it. We will learn of many types in Python but for now we will start with just four:

Types of data

In programming our data comes in different *types*. The type of a particular piece of data is important because it dictates what sort of operations can be applied to it. We will learn of many types in Python but for now we will start with just four:

- Integers — Python type name `int`

Types of data

In programming our data comes in different *types*. The type of a particular piece of data is important because it dictates what sort of operations can be applied to it. We will learn of many types in Python but for now we will start with just four:

- Integers — Python type name `int`
- Real Numbers[1]— Python type name `float` short for *floating point number*

[1] Our `float` type is only an *approximation* of real numbers

Types of data

In programming our data comes in different *types*. The type of a particular piece of data is important because it dictates what sort of operations can be applied to it. We will learn of many types in Python but for now we will start with just four:

- Integers — Python type name `int`
- Real Numbers[1]— Python type name `float` short for *floating point number*
- Text — Python type name `str` short for string. Represents a sequence of characters.

[1] Our `float` type is only an *approximation* of real numbers

Examples of literals

As mentioned literals represent literally a value in our source code.

Examples of literals

As mentioned literals represent literally a value in our source code.

- `int` literals appear as integer numbers we're familiar with.
  e.g. `25`, `100`, `-274`

Examples of literals

As mentioned literals represent literally a value in our source code.

- `int` literals appear as integer numbers we're familiar with.
  e.g. `25`, `100`, `-274`
- `float` literals appear as real numbers we're familiar with.
  e.g. `27.11`, `1.137e-2`, `0.1`

Examples of literals

As mentioned literals represent literally a value in our source code.

- `int` literals appear as integer numbers we're familiar with.
  e.g. `25`, `100`, `-274`
- `float` literals appear as real numbers we're familiar with.
  e.g. `27.11`, `1.137e-2`, `0.1`
- `string` literals appear as text enclosed in either double or single quotes.
  e.g. `"Hello!"`, `'this is one string'`.

Examples of literals

As mentioned literals represent literally a value in our source code.

- `int` literals appear as integer numbers we're familiar with.
  e.g. `25`, `100`, `-274`
- `float` literals appear as real numbers we're familiar with.
  e.g. `27.11`, `1.137e-2`, `0.1`
- `string` literals appear as text enclosed in either double or single quotes.
  e.g. `"Hello!"`, `'this is one string'`.

Since a literal is one of our atomic expressions, each of these examples is also an example of an expression! The value these expressions evaluate to are their literal value.

Identifiers

Our other type of atomic expression is *identifiers*. In programming languages identifiers are our free variables that can be bound to different values, like the names or parameters of our functions.

Identifiers

Our other type of atomic expression is *identifiers*. In programming languages identifiers are our free variables that can be bound to different values, like the names or parameters of our functions.

Each language has its own rules for what constitutes a valid identifier. In Python the rules are quite simple.

Rules of Python identifiers

A valid Python identifier meets the following three rules:

Rules of Python identifiers

A valid Python identifier meets the following three rules:

- Comprises only alphanumeric characters and underscores

Rules of Python identifiers

A valid Python identifier meets the following three rules:

- Comprises only alphanumeric characters and underscores
- Begins with an alphabetic character or an underscore

Rules of Python identifiers

A valid Python identifier meets the following three rules:

- Comprises only alphanumeric characters and underscores
- Begins with an alphabetic character or an underscore
- Cannot be a *keyword*

Rules of Python identifiers

A valid Python identifier meets the following three rules:

- Comprises only alphanumeric characters and underscores
- Begins with an alphabetic character or an underscore
- Cannot be a *keyword*

Keywords are reserved words that have special meaning in a programming language, and as such are reserved for that purpose. We will see some Python keywords shortly.

Binary operator expressions

Our next type of expression we'll discuss is that of a binary operator expression. This is an expression that uses a binary operator such as addition, subtraction, multiplication, division, or exponentiation for example.

Binary operator expressions

Our next type of expression we'll discuss is that of a binary operator expression. This is an expression that uses a binary operator such as addition, subtraction, multiplication, division, or exponentiation for example.

Our tokens for these operators are `+`, `-`, `*`, `/`, `**` respectively.

Binary operator expressions

Our next type of expression we'll discuss is that of a binary operator expression. This is an expression that uses a binary operator such as addition, subtraction, multiplication, division, or exponentiation for example.

Our tokens for these operators are `+`, `-`, `*`, `/`, `**` respectively.

So a binary operator expression takes the form of `lhs operator rhs`. What can we substitute for `lhs` or `rhs` though?

Thinking about algebraic expressions

How are algebraic expressions themselves really constructed?
Forgetting Python, what can appear as the operands of an addition operator in an algebraic expression?

Thinking about algebraic expressions

How are algebraic expressions themselves really constructed?
Forgetting Python, what can appear as the operands of an addition operator in an algebraic expression?

- Just numbers? e.g. $3 + 5$

Thinking about algebraic expressions

How are algebraic expressions themselves really constructed?
Forgetting Python, what can appear as the operands of an addition operator in an algebraic expression?

- Just numbers? e.g. $3 + 5$
  - Counter example: $x + 5$

Thinking about algebraic expressions

How are algebraic expressions themselves really constructed?

Forgetting Python, what can appear as the operands of an addition operator in an algebraic expression?

- Just numbers? e.g. $3 + 5$
  - Counter example: $x + 5$
- Only numbers and variables?

Thinking about algebraic expressions

How are algebraic expressions themselves really constructed? Forgetting Python, what can appear as the operands of an addition operator in an algebraic expression?

- Just numbers? e.g. $3 + 5$
  - Counter example: $x + 5$
- Only numbers and variables?
  - Counter example: $3 * 2 + 7/2$

Thinking about algebraic expressions

How are algebraic expressions themselves really constructed? Forgetting Python, what can appear as the operands of an addition operator in an algebraic expression?

- Just numbers? e.g. $3 + 5$
  - Counter example: $x + 5$
- Only numbers and variables?
  - Counter example: $3 * 2 + 7/2$
- The operands of our arithmetic operators are themselves also expressions! This is true in mathematics as well as in Python

Format of a binary operator expression in Python

expr operator expr

Format of a binary operator expression in Python

These can both be any expression

expr operator expr

Format of a binary operator expression in Python

These can both be any expression

expr operator expr

Any binary operator

Parenthetical expressions

Considering our expression $3 * 2 + 7/2$ it must mean that $3 * 2$ is the left-hand operand of the addition operator and that $7/2$ is the right-hand operator.

Parenthetical expressions

Considering our expression $3 * 2 + 7/2$ it must mean that $3 * 2$ is the left-hand operand of the addition operator and that $7/2$ is the right-hand operator.

We know this because of precedence of arithmetic operators. But what if we wanted to perform addition specifically on the 3 and 2?

Parenthetical expressions

Considering our expression $3 * 2 + 7/2$ it must mean that $3 * 2$ is the left-hand operand of the addition operator and that $7/2$ is the right-hand operator.

We know this because of precedence of arithmetic operators. But what if we wanted to perform addition specifically on the 3 and 2?

In arithmetic we'd use parentheses to enforce precedence. In Python we can do the same.

Parenthetical expressions

Considering our expression $3 * 2 + 7/2$ it must mean that $3 * 2$ is the left-hand operand of the addition operator and that $7/2$ is the right-hand operator.

We know this because of precedence of arithmetic operators. But what if we wanted to perform addition specifically on the 3 and 2?

In arithmetic we'd use parentheses to enforce precedence. In Python we can do the same.

So a valid expression in Python is also `(expr)`, where `expr` can be replaced with any of our expressions.

Unary operator expressions

We have some unary operators in Python, namely +, -, and   .

Unary operator expressions

We have some unary operators in Python, namely +, −, and .

Much like our binary operators the operands of these operators are also expressions, the only difference is they only have one. The unary operators have their operand appear to the right of the operator.

Unary operator expressions

We have some unary operators in Python, namely +, -, and .

Much like our binary operators the operands of these operators are also expressions, the only difference is they only have one. The unary operators have their operand appear to the right of the operator.

e.g. -(3+2)

Example of an expression so far

$$-(3+2)*5-10/2$$

To break this down into its components we have to start from the *last* operator to take place. That is whatever is the lowest precedence, then repeat.

Example of an expression so far

$$-(3+2)*5-10/2$$

- So, we have `lhs BINARYMINUS rhs`

Example of an expression so far

$$-(3+2)*5-10/2$$

- So, we have `lhs BINARYMINUS rhs`
- Our `lhs` is the entire expression `-(3+2)`

Example of an expression so far

$$-(3+2)*5-10/2$$

- So, we have `lhs BINARYMINUS rhs`
- Our `lhs` is the entire expression `-(3+2)`
    - So our `lhs` then is `UNARYMINUS uexpr`

Example of an expression so far

$$-(3+2)*5-10/2$$

- So, we have `lhs BINARYMINUS rhs`
- Our `lhs` is the entire expression `-(3+2)`
    - So our `lhs` then is `UNARYMINUS uexpr`
    - Our `uexpr` here is `(pexpr)`

Example of an expression so far

$$-(3+2) * 5 - 10/2$$

- So, we have `lhs BINARYMINUS rhs`
- Our `lhs` is the entire expression `-(3+2)`
    - So our `lhs` then is `UNARYMINUS uexpr`
    - Our `uexpr` here is `(pexpr)`
        - Our `pexpr` is itself a binary expression, so it is `lhs BINARYPLUS rhs`

Example of an expression so far

$$-(3+2)*5-10/2$$

- So, we have `lhs BINARYMINUS rhs`
- Our `lhs` is the entire expression `-(3+2)`
  - So our `lhs` then is `UNARYMINUS uexpr`
  - Our `uexpr` here is `(pexpr)`
    - Our `pexpr` is itself a binary expression, so it is `lhs BINARYPLUS rhs`
    - In this case, our `lhs` and `rhs` are finally atomics, the literals 3 and 2 respectively.

Example of an expression so far

$$-(3+2)*5-10/2$$

- So, we have `lhs BINARYMINUS rhs`
- Our `lhs` is the entire expression `-(3+2)`
  - So our `lhs` then is `UNARYMINUS uexpr`
  - Our `uexpr` here is `(pexpr)`
    - Our `pexpr` is itself a binary expression, so it is `lhs BINARYPLUS rhs`
    - In this case, our `lhs` and `rhs` are finally atomics, the literals 3 and 2 respectively.
- Our `rhs` is the expression `lhs DIVISION 2`

Example of an expression so far

$$-(3+2)*5-10/2$$

- So, we have `lhs BINARYMINUS rhs`
- Our `lhs` is the entire expression `-(3+2)`
  - So our `lhs` then is `UNARYMINUS uexpr`
  - Our `uexpr` here is `(pexpr)`
    - Our `pexpr` is itself a binary expression, so it is `lhs BINARYPLUS rhs`
    - In this case, our `lhs` and `rhs` are finally atomics, the literals 3 and 2 respectively.
- Our `rhs` is the expression `lhs DIVISION 2`
  - Here, again, our operands are literals (10 and 2 respectively), so we are finally done.

Example of an expression so far

$$-(3+2) * 5 - 10/2$$

Any non-atomic expressions is made up of sub expressions!

Example of an expression so far

$$-(3+2)*5-10/2$$

Any non-atomic expressions is made up of sub expressions!

As we just saw, even this relatively small expression is fairly complex in how many expressions it comprises.

A note on expressions and types

In Python the resultant type of the evaluation of an expression depends on the expression. One place we can see this occur is in arithmetic in Python.

$$3 * 5 \longrightarrow 15$$
$$3 * 2.0 \longrightarrow 10.0$$
$$10/2 \longrightarrow 5.0$$

The type of the operands can affect the behaviour of an operator. The multiplication operator is shown to produce an integer when multiplying two integers. However, when multiplying an integer by a float the result is a float. This is also true of addition and subtraction.

Division, however, always produces a float.

Function call expression

Another way we just discussed producing values was with our function applications.

Function call expression

Another way we just discussed producing values was with our function applications.

A function application[2] is also an expression, as it evaluates to a final value.

[2] Called a function call in Python, and most programming languages.

Function call expression format

`expr(argList)`

Function call expression format

`expr(argList)`

Expression that evaluates to a function
typically an identifier that is its name

Function call expression format

Series of expressions separated by commas

`expr(argList)`

Expression that evaluates to a function
typically an identifier that is its name

Example function call

```
myFn(3+2, 27.58*3.14*1e-5, "hello")
```

Table of Contents

Operations on strings

We've been presenting things much from the perspective of arithmetic expressions.

Operations on strings

We've been presenting things much from the perspective of arithmetic expressions.

However, given the existence of our `str` type we must have other types of expressions.

A note on sequences

A `str` in Python is an ordered collection of characters. We will also learn about many other ordered collections in the future.

For discussing behaviours, it is often helpful to group together similar data types into a general classification. The general classification of most of the ordered collection data types we'll discuss belong to what we'll call a *sequence*.

The operations we are going to now discuss that work on strings work on any sequence.

Concatenation

Concatenation is simply the joining of two strings by adding one string to the end of another to construct a new string.

Concatenation

Concatenation is simply the joining of two strings by adding one string to the end of another to construct a new string.

The addition operator + is also *overloaded* as the concatenation operator. Which operator it is depends on the context of its operands.

Concatenation

Concatenation is simply the joining of two strings by adding one string to the end of another to construct a new string.

The addition operator + is also *overloaded* as the concatenation operator. Which operator it is depends on the context of its operands.

The expression `"cave"+"man"` yields the value `"caveman"`.

Repetition

A string can be multiplied by an integer value to produce a new string which is the original string operand repeated a number of times equal to the integer operand.

Repetition

A string can be multiplied by an integer value to produce a new string which is the original string operand repeated a number of times equal to the integer operand.

The expression `"abc"*3` yields the value `"abcabcabc"`.

Indexing

The index operator `[]` is a binary operator that takes the form of `lhs[rhs]`.

Indexing

The index operator `[]` is a binary operator that takes the form of `lhs[rhs]`.

The `lhs` expression must evaluate to a sequence. The `rhs` expression must evaluate to an integer.

Indexing

The index operator `[]` is a binary operator that takes the form of `lhs[rhs]`.

The `lhs` expression must evaluate to a sequence. The `rhs` expression must evaluate to an integer.

The resultant value is the item in the sequence at the position indicated by the integer.

Indexing behaviour in Python

$$\text{“}\mathit{Oilers}\text{”}\,[2] \longrightarrow \text{“}\mathit{l}\text{”}$$

Indexing behaviour in Python

$$\text{``}\mathit{Oilers}\text{''}\,[2] \longrightarrow \text{``}\mathit{l}\text{''}$$

Why does `"Oilers"[2]"` not produce `"i"`, as `"i"` is the second character in the string?

Indexing behaviour in Python

$$\text{“}\mathit{Oilers}\text{”}[2] \longrightarrow \text{“}\mathit{l}\text{”}$$

Why does `"Oilers"[2]` not produce `"i"`, as `"i"` is the second character in the string?

In Python, as well as many programming languages, the positions of our collections begins with *zero*. We call this *zero indexing*.

Indexing behaviour in Python

$$\text{“}\textit{Oilers}\text{”}[2] \longrightarrow \text{“}\textit{l}\text{”}$$

Why does `"Oilers"[2]` not produce `"i"`, as `"i"` is the second character in the string?

In Python, as well as many programming languages, the positions of our collections begins with *zero*. We call this *zero indexing*.

In order for an indexing expression to be valid, the index must correspond to a valid position within the sequence. The valid positions go from 0 to $n - 1$ where n is the *length* of the sequence.

Indexing behaviour in Python

$$\text{“}\textit{Oilers}\text{”}[0] \longrightarrow \text{“}\textit{O}\text{”}$$

$$\text{“}\textit{Oilers}\text{”}[1] \longrightarrow \text{“}\textit{i}\text{”}$$

$$\text{“}\textit{Oilers}\text{”}[2] \longrightarrow \text{“}\textit{l}\text{”}$$

$$\text{“}\textit{Oilers}\text{”}[3] \longrightarrow \text{“}\textit{e}\text{”}$$

$$\text{“}\textit{Oilers}\text{”}[4] \longrightarrow \text{“}\textit{r}\text{”}$$

$$\text{“}\textit{Oilers}\text{”}[5] \longrightarrow \text{“}\textit{s}\text{”}$$

Indexing behaviour in Python

$$\text{"}\mathit{Oilers}\text{"}\,[0] \longrightarrow \text{"}\mathit{O}\text{"}$$

$$\text{"}\mathit{Oilers}\text{"}\,[1] \longrightarrow \text{"}\mathit{i}\text{"}$$

$$\text{"}\mathit{Oilers}\text{"}\,[2] \longrightarrow \text{"}\mathit{l}\text{"}$$

$$\text{"}\mathit{Oilers}\text{"}\,[3] \longrightarrow \text{"}\mathit{e}\text{"}$$

$$\text{"}\mathit{Oilers}\text{"}\,[4] \longrightarrow \text{"}\mathit{r}\text{"}$$

$$\text{"}\mathit{Oilers}\text{"}\,[5] \longrightarrow \text{"}\mathit{s}\text{"}$$

Knowledge Check: What is the result of the expression `("abc"[1]+"hello")[3]`?

Slicing expressions

The last operation we'll discuss for now on sequence is that of *slicing*. A slicing expression allows us to produce a subsequence of the given sequence, and there are two forms of it.

Slicing expressions

The last operation we'll discuss for now on sequence is that of *slicing*. A slicing expression allows us to produce a subsequence of the given sequence, and there are two forms of it.

- `s[i:j]`

Slicing expressions

The last operation we'll discuss for now on sequence is that of *slicing*. A slicing expression allows us to produce a subsequence of the given sequence, and there are two forms of it.

- `s[i:j]`
  - `s` — an expression that evaluates to a sequence

Slicing expressions

The last operation we'll discuss for now on sequence is that of *slicing*. A slicing expression allows us to produce a subsequence of the given sequence, and there are two forms of it.

- `s[i:j]`
    - `s` — an expression that evaluates to a sequence
    - `i` and `j` — expressions that evaluate to an integer

Slicing expressions

The last operation we'll discuss for now on sequence is that of *slicing*. A slicing expression allows us to produce a subsequence of the given sequence, and there are two forms of it.

- `s[i:j]` — produces the sequence that includes the items from `s` in the range of indices $[i, j)$.
  - `s` — an expression that evaluates to a sequence
  - `i` and `j` — expressions that evaluate to an integer

Slicing expressions

The last operation we'll discuss for now on sequence is that of *slicing*. A slicing expression allows us to produce a subsequence of the given sequence, and there are two forms of it.

- `s[i:j]` — produces the sequence that includes the items from `s` in the range of indices $[i, j)$.
  - `s` — an expression that evaluates to a sequence
  - `i` and `j` — expressions that evaluate to an integer
- `s[i:j:k]`

Slicing expressions

The last operation we'll discuss for now on sequence is that of *slicing*. A slicing expression allows us to produce a subsequence of the given sequence, and there are two forms of it.

- `s[i:j]` — produces the sequence that includes the items from `s` in the range of indices $[i, j)$.
  - `s` — an expression that evaluates to a sequence
  - `i` and `j` — expressions that evaluate to an integer
- `s[i:j:k]`
  - `s`, `i`, and `j` — same as above

Slicing expressions

The last operation we'll discuss for now on sequence is that of *slicing*. A slicing expression allows us to produce a subsequence of the given sequence, and there are two forms of it.

- `s[i:j]` — produces the sequence that includes the items from `s` in the range of indices $[i, j)$.
  - `s` — an expression that evaluates to a sequence
  - `i` and `j` — expressions that evaluate to an integer
- `s[i:j:k]`
  - `s`, `i`, and `j` — same as above
  - `k` — an expression that evaluate to an integer

Slicing expressions

The last operation we'll discuss for now on sequence is that of *slicing*. A slicing expression allows us to produce a subsequence of the given sequence, and there are two forms of it.

- `s[i:j]` — produces the sequence that includes the items from `s` in the range of indices $[i, j)$.
  - `s` — an expression that evaluates to a sequence
  - `i` and `j` — expressions that evaluate to an integer
- `s[i:j:k]` — produces the sequence that includes the items from `s` in the range of indices $[i, j)$ stepping by `k` positions.
  - `s`, `i`, and `j` — same as above
  - `k` — an expression that evaluate to an integer

Slicing expressions

The last operation we'll discuss for now on sequence is that of *slicing*. A slicing expression allows us to produce a subsequence of the given sequence, and there are two forms of it.

- `s[i:j]` — produces the sequence that includes the items from `s` in the range of indices $[i, j)$.
    - `s` — an expression that evaluates to a sequence
    - `i` and `j` — expressions that evaluate to an integer
- `s[i:j:k]` — produces the sequence that includes the items from `s` in the range of indices $[i, j)$ stepping by `k` positions.
    - `s`, `i`, and `j` — same as above
    - `k` — an expression that evaluate to an integer
- Omitting either `i` or `j` will replace them with either the beginning or end of the string depending on context.

Slicing examples

$$\texttt{"CMPUT 274 is fun"[4:9]} \longrightarrow \texttt{"T 274"}$$

$$\texttt{"CMPUT 274 is fun"[10:16]} \longrightarrow \texttt{"is fun"}$$

$$\texttt{"CMPUT 274 is fun"[0:16:2]} \longrightarrow \texttt{"CPT24i u"}$$

$$\texttt{"CMPUT 274 is fun"[1:16:2]} \longrightarrow \texttt{"MU 7 sfn"}$$

$$\texttt{"CMPUT 274 is fun"[1:]} \longrightarrow \texttt{"MPUT 274 is fun"}$$

$$\texttt{"CMPUT 274 is fun"[16::-1]} \longrightarrow \texttt{"nuf si 472 TUPMC"}$$

Table of Contents

Keyword list

We mentioned earlier that an identifier could not be a keyword, as keywords are reserved in Python and each have their own special meaning.

We now present a list of the keywords, however we will not elaborate on all of them now. Rather we will define keywords purposes as they become relevant.

False	None	True	and	as	assert	break
class	continue	def	del	elif	else	except
finally	for	from	global	if	import	in
is	lambda	nonlocal	not	or	pass	raise
return	try	while	with	yield		

Keyword meanings

Each keyword has its own special meaning, there is no general rule of thumb that dictates what keywords do.

Some keywords act as literals, `True`, `False`, and `None` are some such examples.

Some keywords define language features for us.

Some keywords even work as operators...