Built-in Types of Data
Outline

1. Types
2. Definitions
3. Strings
4. Integers
5. Floating-point Numbers
6. Booleans
7. Functions and APIs
8. Type Conversion
9. Interactive Python
Types

A data type is set of values and a set of operations defined on those values.

Python supports several built-in data types: `int` (for integers), `float` (for floating-point numbers), `str` (for sequences of characters), `bool` (for true/false values), and others.

Python also allows us to compose our own data types, i.e., it supports object-oriented programming (OOP).
A data type is set of values and a set of operations defined on those values

Python supports several built-in data types: `int` (for integers), `float` (for floating-point numbers), `str` (for sequences of characters), `bool` (for true/false values), and others

Python also allows us to compose our own data types, i.e., it supports object-oriented programming (OOP)
Types

A data type is a set of values and a set of operations defined on those values.

Python supports several built-in data types: `int` (for integers), `float` (for floating-point numbers), `str` (for sequences of characters), `bool` (for true/false values), and others.

Python also allows us to compose our own data types, i.e., it supports object-oriented programming (OOP).
A data type is set of values and a set of operations defined on those values

Python supports several built-in data types: `int` (for integers), `float` (for floating-point numbers), `str` (for sequences of characters), `bool` (for true/false values), and others

Python also allows us to compose our own data types, ie, it supports object-oriented programming (OOP)
A literal is a Python-code representation of a data-type value

For example, 1234 and 99 are int literals; 3.14159 and 2.71828 are float literals; True and False are bool literals; 'Hello, World' is a str literal

An operator is a Python-code representation of a data-type operation

For example, + and * represent addition and multiplication for integers and floating-point numbers; and, or, and not represent boolean operations

An identifier is a Python-code representation of a name

Each identifier is a sequence of letters, digits, and underscores, the first of which is not a digit

For example, abc, Ab_, abc123, and a_b are valid identifiers, but Ab*, 1abc, and a+b are not

Certain keywords, such as and, import, in, def, while, from, and lambda, are reserved, and we cannot use them as identifiers; others such as int, sum, min, max, len, id, file, and input, have special meaning, so it is best not to use them, either
A literal is a Python-code representation of a data-type value

For example, 1234 and 99 are int literals; 3.14159 and 2.71828 are float literals; True and False are bool literals; 'Hello, World' is a str literal

An operator is a Python-code representation of a data-type operation

For example, + and * represent addition and multiplication for integers and floating-point numbers; and, or, and not represent boolean operations

An identifier is a Python-code representation of a name

Each identifier is a sequence of letters, digits, and underscores, the first of which is not a digit

For example, abc, Ab_, abc123, and a_b are valid identifiers, but Ab*, 1abc, and a+b are not

Certain keywords, such as and, import, in, def, while, from, and lambda, are reserved, and we cannot use them as identifiers; others such as int, sum, min, max, len, id, file, and input, have special meaning, so it is best not to use them, either
Definitions

A literal is a Python-code representation of a data-type value

For example, 1234 and 99 are int literals; 3.14159 and 2.71828 are float literals; True and False are bool literals; ’Hello, World’ is a str literal

An operator is a Python-code representation of a data-type operation

For example, + and * represent addition and multiplication for integers and floating-point numbers; and, or, and not represent boolean operations

An identifier is a Python-code representation of a name

Each identifier is a sequence of letters, digits, and underscores, the first of which is not a digit

For example, abc, Ab_, abc123, and a_b are valid identifiers, but Ab*, 1abc, and a+b are not

Certain keywords, such as and, import, in, def, while, from, and lambda, are reserved, and we cannot use them as identifiers; others such as int, sum, min, max, len, id, file, and input, have special meaning, so it is best not to use them, either
Definitions

A literal is a Python-code representation of a data-type value

For example, 1234 and 99 are int literals; 3.14159 and 2.71828 are float literals; True and False are bool literals; 'Hello, World' is a str literal

An operator is a Python-code representation of a data-type operation

For example, + and * represent addition and multiplication for integers and floating-point numbers; and, or, and not represent boolean operations

An identifier is a Python-code representation of a name

Each identifier is a sequence of letters, digits, and underscores, the first of which is not a digit

For example, abc, Ab_, abc123, and a_b are valid identifiers, but Ab*, 1abc, and a+b are not

Certain keywords, such as and, import, in, def, while, from, and lambda, are reserved, and we cannot use them as identifiers; others such as int, sum, min, max, len, id, file, and input, have special meaning, so it is best not to use them, either
Definitions

A literal is a Python-code representation of a data-type value

For example, 1234 and 99 are int literals; 3.14159 and 2.71828 are float literals; True and False are bool literals; 'Hello, World' is a str literal

An operator is a Python-code representation of a data-type operation

For example, + and * represent addition and multiplication for integers and floating-point numbers; and, or, and not represent boolean operations

An identifier is a Python-code representation of a name

Each identifier is a sequence of letters, digits, and underscores, the first of which is not a digit

For example, abc, Ab_, abc123, and a_b are valid identifiers, but Ab*, 1abc, and a+b are not

Certain keywords, such as and, import, in, def, while, from, and lambda, are reserved, and we cannot use them as identifiers; others such as int, sum, min, max, len, id, file, and input, have special meaning, so it is best not to use them, either
Definitions

A literal is a Python-code representation of a data-type value

For example, 1234 and 99 are int literals; 3.14159 and 2.71828 are float literals; True and False are bool literals; ’Hello, World’ is a str literal

An operator is a Python-code representation of a data-type operation

For example, + and * represent addition and multiplication for integers and floating-point numbers; and, or, and not represent boolean operations

An identifier is a Python-code representation of a name

Each identifier is a sequence of letters, digits, and underscores, the first of which is not a digit

For example, abc, Ab_, abc123, and a_b are valid identifiers, but Ab*, 1abc, and a+b are not

Certain keywords, such as and, import, in, def, while, from, and lambda, are reserved, and we cannot use them as identifiers; others such as int, sum, min, max, len, id, file, and input, have special meaning, so it is best not to use them, either
Definitions

A literal is a Python-code representation of a data-type value

For example, 1234 and 99 are int literals; 3.14159 and 2.71828 are float literals; True and False are bool literals; ’Hello, World’ is a str literal

An operator is a Python-code representation of a data-type operation

For example, + and * represent addition and multiplication for integers and floating-point numbers; and, or, and not represent boolean operations

An identifier is a Python-code representation of a name

Each identifier is a sequence of letters, digits, and underscores, the first of which is not a digit

For example, abc, Ab_, abc123, and a_b are valid identifiers, but Ab*, 1abc, and a+b are not

Certain keywords, such as and, import, in, def, while, from, and lambda, are reserved, and we cannot use them as identifiers; others such as int, sum, min, max, len, id, file, and input, have special meaning, so it is best not to use them, either
Definitions

A literal is a Python-code representation of a data-type value

For example, 1234 and 99 are int literals; 3.14159 and 2.71828 are float literals; True and False are bool literals; 'Hello, World' is a str literal

An operator is a Python-code representation of a data-type operation

For example, + and * represent addition and multiplication for integers and floating-point numbers; and, or, and not represent boolean operations

An identifier is a Python-code representation of a name

Each identifier is a sequence of letters, digits, and underscores, the first of which is not a digit

For example, abc, Ab_, abc123, and a_b are valid identifiers, but Ab*, 1abc, and a+b are not

Certain keywords, such as and, import, in, def, while, from, and lambda, are reserved, and we cannot use them as identifiers; others such as int, sum, min, max, len, id, file, and input, have special meaning, so it is best not to use them, either
Definitions

A literal is a Python-code representation of a data-type value

For example, 1234 and 99 are int literals; 3.14159 and 2.71828 are float literals; True and False are bool literals; 'Hello, World' is a str literal

An operator is a Python-code representation of a data-type operation

For example, + and * represent addition and multiplication for integers and floating-point numbers; and, or, and not represent boolean operations

An identifier is a Python-code representation of a name

Each identifier is a sequence of letters, digits, and underscores, the first of which is not a digit

For example, abc, Ab_, abc123, and a_b are valid identifiers, but Ab*, 1abc, and a+b are not

Certain keywords, such as and, import, in, def, while, from, and lambda, are reserved, and we cannot use them as identifiers; others such as int, sum, min, max, len, id, file, and input, have special meaning, so it is best not to use them, either
Definitions

A variable is a name associated with a data-type value

For example, the variable `total` might represent the running total of a sequence of numbers

A constant variable describes a variable whose associated data-type value does not change during the execution of a program

For example, the variable `SPEED_OF_LIGHT` might represent the known speed of light

An expression is a combination of literals, variables, and operators that Python evaluates to produce a value

For example, `4 * (x - 3)` is an expression

Python has a natural and well-defined set of precedence rules that fully specify the order in which the operators are applied in an expression

- For arithmetic operations, multiplication and division are performed before addition and subtraction
- When arithmetic operations have the same precedence, they are left associative, with the exception of the exponentiation operator `**`, which is right associative
- We can use parentheses to override precedence rules
Definitions

A variable is a name associated with a data-type value

For example, the variable `total` might represent the running total of a sequence of numbers.

A constant variable describes a variable whose associated data-type value does not change during the execution of a program.

For example, the variable `SPEED_OF_LIGHT` might represent the known speed of light.

An expression is a combination of literals, variables, and operators that Python evaluates to produce a value.

For example, `4 * (x - 3)` is an expression.

Python has a natural and well-defined set of precedence rules that fully specify the order in which the operators are applied in an expression:

- For arithmetic operations, multiplication and division are performed before addition and subtraction.
- When arithmetic operations have the same precedence, they are left associative, with the exception of the exponentiation operator `**`, which is right associative.
- We can use parentheses to override precedence rules.
Definitions

A variable is a name associated with a data-type value

For example, the variable `total` might represent the running total of a sequence of numbers

A constant variable describes a variable whose associated data-type value does not change during the execution of a program

For example, the variable `SPEED_OF_LIGHT` might represent the known speed of light

An expression is a combination of literals, variables, and operators that Python evaluates to produce a value

For example, `4 * (x - 3)` is an expression

Python has a natural and well-defined set of precedence rules that fully specify the order in which the operators are applied in an expression

- For arithmetic operations, multiplication and division are performed before addition and subtraction
- When arithmetic operations have the same precedence, they are left associative, with the exception of the exponentiation operator `**`, which is right associative
- We can use parentheses to override precedence rules
Definitions

A variable is a name associated with a data-type value

For example, the variable `total` might represent the running total of a sequence of numbers

A constant variable describes a variable whose associated data-type value does not change during the execution of a program

For example, the variable `SPEED_OF_LIGHT` might represent the known speed of light

An expression is a combination of literals, variables, and operators that Python evaluates to produce a value

For example, `4 * (x - 3)` is an expression

Python has a natural and well-defined set of precedence rules that fully specify the order in which the operators are applied in an expression

- For arithmetic operations, multiplication and division are performed before addition and subtraction
- When arithmetic operations have the same precedence, they are left associative, with the exception of the exponentiation operator `**`, which is right associative
- We can use parentheses to override precedence rules
Definitions

A variable is a name associated with a data-type value

For example, the variable `total` might represent the running total of a sequence of numbers.

A constant variable describes a variable whose associated data-type value does not change during the execution of a program.

For example, the variable `SPEED_OF_LIGHT` might represent the known speed of light.

An expression is a combination of literals, variables, and operators that Python evaluates to produce a value.

For example, `4 * (x - 3)` is an expression.

Python has a natural and well-defined set of precedence rules that fully specify the order in which the operators are applied in an expression:

- For arithmetic operations, multiplication and division are performed before addition and subtraction.
- When arithmetic operations have the same precedence, they are left associative, with the exception of the exponentiation operator `**`, which is right associative.
- We can use parentheses to override precedence rules.
Definitions

A variable is a name associated with a data-type value

For example, the variable `total` might represent the running total of a sequence of numbers

A constant variable describes a variable whose associated data-type value does not change during the execution of a program

For example, the variable `SPEED_OF_LIGHT` might represent the known speed of light

An expression is a combination of literals, variables, and operators that Python evaluates to produce a value

For example, `4 * (x - 3)` is an expression

Python has a natural and well-defined set of precedence rules that fully specify the order in which the operators are applied in an expression

- For arithmetic operations, multiplication and division are performed before addition and subtraction
- When arithmetic operations have the same precedence, they are left associative, with the exception of the exponentiation operator `**`, which is right associative
- We can use parentheses to override precedence rules
Definitions

A variable is a name associated with a data-type value

For example, the variable `total` might represent the running total of a sequence of numbers

A constant variable describes a variable whose associated data-type value does not change during the execution of a program

For example, the variable `SPEED_OF_LIGHT` might represent the known speed of light

An expression is a combination of literals, variables, and operators that Python evaluates to produce a value

For example, `4 * (x - 3)` is an expression

Python has a natural and well-defined set of precedence rules that fully specify the order in which the operators are applied in an expression

- For arithmetic operations, multiplication and division are performed before addition and subtraction
- When arithmetic operations have the same precedence, they are left associative, with the exception of the exponentiation operator `**`, which is right associative
- We can use parentheses to override precedence rules
Definitions

A variable is a name associated with a data-type value

For example, the variable `total` might represent the running total of a sequence of numbers

A constant variable describes a variable whose associated data-type value does not change during the execution of a program

For example, the variable `SPEED_OF_LIGHT` might represent the known speed of light

An expression is a combination of literals, variables, and operators that Python evaluates to produce a value

For example, `4 * (x - 3)` is an expression

Python has a natural and well-defined set of precedence rules that fully specify the order in which the operators are applied in an expression

- For arithmetic operations, multiplication and division are performed before addition and subtraction
- When arithmetic operations have the same precedence, they are left associative, with the exception of the exponentiation operator `**`, which is right associative
- We can use parentheses to override precedence rules
Definitions

We use an assignment statement to define a variable and associate it with a data-type value

<variable> = <value>

For example, the statement

a = 1234

defines an identifier a to be a new variable and associates it with the integer data-type value 1234

To represent the absence of a value, we can use the value None

All data values in Python are represented by objects, each characterized by its identity (or memory address), type, and value

For example, the following figure shows how the variable a as defined above, might be represented in memory (left) and conceptually (right)
Definitions

We use an assignment statement to define a variable and associate it with a data-type value

\[
\text{<variable> } = \text{ <value> }
\]

For example, the statement

\[a = 1234\]

defines an identifier \(a\) to be a new variable and associates it with the integer data-type value 1234

To represent the absence of a value, we can use the value \(\text{None}\)

All data values in Python are represented by objects, each characterized by its identity (or memory address), type, and value

For example, the following figure shows how the variable \(a\) as defined above, might be represented in memory (left) and conceptually (right)

![Diagram showing memory representation and conceptual representation of variable a with value 1234]
Definitions

We use an assignment statement to define a variable and associate it with a data-type value

<variable> = <value>

For example, the statement

```python
a = 1234
```

defines an identifier `a` to be a new variable and associates it with the integer data-type value `1234`

To represent the absence of a value, we can use the value `None`

All data values in Python are represented by objects, each characterized by its identity (or memory address), type, and value

For example, the following figure shows how the variable `a` as defined above, might be represented in memory (left) and conceptually (right)
Definitions

We use an assignment statement to define a variable and associate it with a data-type value

<variable> = <value>

For example, the statement

a = 1234

defines an identifier a to be a new variable and associates it with the integer data-type value 1234

To represent the absence of a value, we can use the value None

All data values in Python are represented by objects, each characterized by its identity (or memory address), type, and value

For example, the following figure shows how the variable a as defined above, might be represented in memory (left) and conceptually (right)
Definitions

We use an assignment statement to define a variable and associate it with a data-type value

\[ \text{<variable>} = \text{<value>} \]

For example, the statement

\[ a = 1234 \]

defines an identifier \( a \) to be a new variable and associates it with the integer data-type value \( 1234 \)

To represent the absence of a value, we can use the value \( \text{None} \)

All data values in Python are represented by objects, each characterized by its identity (or memory address), type, and value

For example, the following figure shows how the variable \( a \) as defined above, might be represented in memory (left) and conceptually (right)
Definitions

We use an assignment statement to define a variable and associate it with a data-type value

\[ \text{<variable> = <value>} \]

For example, the statement

\[ a = 1234 \]

defines an identifier \( a \) to be a new variable and associates it with the integer data-type value 1234

To represent the absence of a value, we can use the value \( \text{None} \)

All data values in Python are represented by objects, each characterized by its identity (or memory address), type, and value

For example, the following figure shows how the variable \( a \) as defined above, might be represented in memory (left) and conceptually (right)
Definitions

Consider the following code that exchanges \( a = 1234 \) and \( b = 99 \) (more precisely, the objects bound to \( a \) and \( b \))

\[
\begin{align*}
t &= a \\
a &= b \\
b &= t
\end{align*}
\]

The object-level trace for the code is shown below
Definitions

Consider the following code that exchanges $a = 1234$ and $b = 99$ (more precisely, the objects bound to $a$ and $b$)

```python
t = a
a = b
b = t
```

The object-level trace for the code is shown below
Strings

The `str` data type represents strings (sequences of characters), for use in text processing.

A `str` literal is specified by enclosing a sequence of characters in matching single quotes.

For example, `'ab'` is a `str` literal.

We can specify tab, newline, backslash, and single quote characters using escape sequences `\t`, `\n`, `\'`, and `'\''`, respectively.

We can concatenate two strings using the `+` operator.

For example, the expression `'123' + '456'` evaluates to the `str` object whose value is `'123456'`.

We can multiply a `str` object `s` by a number `n` to obtain a `str` object whose value is the string `s` repeated `n` times.

For example, the expressions `3 * 'ab'` and `'ab' * 3` evaluate to the `str` object whose value is `'ababab'`.

The `str` data type

<table>
<thead>
<tr>
<th>values</th>
<th>sequences of characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>'Hello, World', 'Python's'</td>
</tr>
<tr>
<td>operations</td>
<td>concatenate, multiply</td>
</tr>
<tr>
<td>operators</td>
<td><code>+</code>, <code>*</code></td>
</tr>
</tbody>
</table>
**Strings**

The `str` data type represents strings (sequences of characters), for use in text processing.

A `str` literal is specified by enclosing a sequence of characters in matching single quotes. For example, `'ab'` is a `str` literal.

We can specify tab, newline, backslash, and single quote characters using escape sequences `	`, `
`, `\`, and `'`, respectively.

We can concatenate two strings using the `+` operator. For example, the expression `'123' + '456'` evaluates to the `str` object whose value is `'123456'`.

We can multiply a `str` object `s` by a number `n` to obtain a `str` object whose value is the string `s` repeated `n` times. For example, the expressions `3 * 'ab'` and `'ab' * 3` evaluate to the `str` object whose value is `'ababab'`.

The `str` data type values typical literals `'Hello, World', 'Python\'s'` operations concatenate, multiply operators `+`, `*`
**Strings**

The `str` data type represents strings (sequences of characters), for use in text processing.

A `str` literal is specified by enclosing a sequence of characters in matching single quotes. For example, `'ab'` is a `str` literal.

We can specify tab, newline, backslash, and single quote characters using escape sequences `	`, `
`, `\`, and `'`, respectively.

We can concatenate two strings using the `+` operator. For example, the expression `'123' + '456'` evaluates to the `str` object whose value is `'123456'`.

We can multiply a `str` object `s` by a number `n` to obtain a `str` object whose value is the string `s` repeated `n` times. For example, the expressions `3 * 'ab'` and `'ab' * 3` evaluate to the `str` object whose value is `'ababab'`.

The `str` data type:

<table>
<thead>
<tr>
<th>values</th>
<th>sequences of characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>'Hello, World', 'Python\’s'</td>
</tr>
<tr>
<td>operations</td>
<td>concatenate, multiply</td>
</tr>
<tr>
<td>operators</td>
<td>+, *</td>
</tr>
</tbody>
</table>
Strings

The `str` data type represents strings (sequences of characters), for use in text processing.

A `str` literal is specified by enclosing a sequence of characters in matching single quotes.

For example, `'ab'` is a `str` literal.

We can specify tab, newline, backslash, and single quote characters using escape sequences `'\t'`, `'\n'`, `'\'`, and `'\''`, respectively.

We can concatenate two strings using the `+` operator.

For example, the expression `'123' + '456'` evaluates to the `str` object whose value is `'123456'`.

We can multiply a `str` object `s` by a number `n` to obtain a `str` object whose value is the string `s` repeated `n` times.

For example, the expressions `3 * 'ab'` and `'ab' * 3` evaluate to the `str` object whose value is `'ababab'`.

The `str` data type.

<table>
<thead>
<tr>
<th>Typical literals</th>
<th>Sequences of characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td><code>Hello, World</code>, <code>Python's</code></td>
</tr>
<tr>
<td>operations</td>
<td>concatenate, multiply</td>
</tr>
<tr>
<td>operators</td>
<td><code>+</code>, <code>*</code></td>
</tr>
</tbody>
</table>
Strings

The `str` data type represents strings (sequences of characters), for use in text processing.

A `str` literal is specified by enclosing a sequence of characters in matching single quotes.

For example, `'ab'` is a `str` literal.

We can specify tab, newline, backslash, and single quote characters using escape sequences `\t`, `\n`, `\'`, and `"`, respectively.

We can concatenate two strings using the `+` operator.

For example, the expression `'123' + '456'` evaluates to the `str` object whose value is `'123456'`.

We can multiply a `str` object `s` by a number `n` to obtain a `str` object whose value is the string `s` repeated `n` times.

For example, the expressions `3 * 'ab'` and `'ab' * 3` evaluate to the `str` object whose value is `'ababab'`.

The `str` data type:

<table>
<thead>
<tr>
<th>values</th>
<th>sequences of characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>'Hello, World', 'Python's'</td>
</tr>
<tr>
<td>operations</td>
<td>concatenate, multiply</td>
</tr>
<tr>
<td>operators</td>
<td>+, *</td>
</tr>
</tbody>
</table>
Strings
The `str` data type represents strings (sequences of characters), for use in text processing.

A `str` literal is specified by enclosing a sequence of characters in matching single quotes.

For example, `'ab'` is a `str` literal.

We can specify tab, newline, backslash, and single quote characters using escape sequences `'	'`, `'
'`, `'\'`, and `'\''`, respectively.

We can concatenate two strings using the `+` operator.

For example, the expression `'123' + '456'` evaluates to the `str` object whose value is `'123456'`.

We can multiply a `str` object `s` by a number `n` to obtain a `str` object whose value is the string `s` repeated `n` times.

For example, the expressions `3 * 'ab'` and `'ab' * 3` evaluate to the `str` object whose value is `'ababab'`.

The `str` data type:

<table>
<thead>
<tr>
<th>values</th>
<th>sequences of characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>`Hello, World', 'Python's'</td>
</tr>
<tr>
<td>operations</td>
<td>concatenate, multiply</td>
</tr>
<tr>
<td>operators</td>
<td><code>+</code>, <code>*</code></td>
</tr>
</tbody>
</table>
Strings

The `str` data type represents strings (sequences of characters), for use in text processing.

A `str` literal is specified by enclosing a sequence of characters in matching single quotes. For example, `'ab'` is a `str` literal.

We can specify tab, newline, backslash, and single quote characters using escape sequences `\t`, `\n`, `\'`, and `'\'', respectively.

We can concatenate two strings using the `+` operator. For example, the expression `'123' + '456'` evaluates to the `str` object whose value is `'123456'`.

We can multiply a `str` object `s` by a number `n` to obtain a `str` object whose value is the string `s` repeated `n` times.

For example, the expressions `3 * 'ab'` and `'ab' * 3` evaluate to the `str` object whose value is `'ababab'`.

The `str` data type

<table>
<thead>
<tr>
<th>values</th>
<th>sequences of characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>'Hello, World', 'Python\’s'</td>
</tr>
<tr>
<td>operations</td>
<td>concatenate, multiply</td>
</tr>
<tr>
<td>operators</td>
<td>+, *</td>
</tr>
</tbody>
</table>
## Strings

The `str` data type represents strings (sequences of characters), for use in text processing.

A `str` literal is specified by enclosing a sequence of characters in matching single quotes.

For example, `'ab'` is a `str` literal.

We can specify tab, newline, backslash, and single quote characters using escape sequences `\t`, `\n`, `\`", and `\''`, respectively.

We can concatenate two strings using the `+` operator.

For example, the expression `'123' + '456'` evaluates to the `str` object whose value is `'123456'`.

We can multiply a `str` object `s` by a number `n` to obtain a `str` object whose value is the string `s` repeated `n` times.

For example, the expressions `3 * 'ab'` and `'ab' * 3` evaluate to the `str` object whose value is `'ababab'`.

The `str` data type

<table>
<thead>
<tr>
<th>values</th>
<th>sequences of characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>'Hello, World', 'Python's'</td>
</tr>
<tr>
<td>operations</td>
<td>concatenate, multiply</td>
</tr>
<tr>
<td>operators</td>
<td>+, *</td>
</tr>
</tbody>
</table>
Strings

The `str` data type represents strings (sequences of characters), for use in text processing.

A `str` literal is specified by enclosing a sequence of characters in matching single quotes.

For example, `'ab'` is a `str` literal.

We can specify tab, newline, backslash, and single quote characters using escape sequences `'\t'`, `'\n'`, `'\'`, and `'\'`, respectively.

We can concatenate two strings using the `+` operator.

For example, the expression `'123' + '456'` evaluates to the `str` object whose value is `'123456'`.

We can multiply a `str` object `s` by a number `n` to obtain a `str` object whose value is the string `s` repeated `n` times.

For example, the expressions `3 * 'ab'` and `'ab' * 3` evaluate to the `str` object whose value is `'ababab'`.

<table>
<thead>
<tr>
<th>The <code>str</code> data type</th>
<th><code>str</code> values</th>
<th><code>str</code> operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typical literals</td>
<td>'Hello, World', 'Python's'</td>
<td></td>
</tr>
<tr>
<td>Operations</td>
<td>+, *</td>
<td></td>
</tr>
</tbody>
</table>
Strings

The `str` data type represents strings (sequences of characters), for use in text processing.

A `str` literal is specified by enclosing a sequence of characters in matching single quotes. For example, `'ab'` is a `str` literal.

We can specify tab, newline, backslash, and single quote characters using escape sequences `'\t'`, `'\n'`, `'\'`, and `'\''`, respectively.

We can concatenate two strings using the `+` operator. For example, the expression `'123' + '456'` evaluates to the `str` object whose value is `'123456'`.

We can multiply a `str` object `s` by a number `n` to obtain a `str` object whose value is the string `s` repeated `n` times.

For example, the expressions `3 * 'ab'` and `'ab' * 3` evaluate to the `str` object whose value is `'ababab'`.

The `str` data type

<table>
<thead>
<tr>
<th>values</th>
<th>sequences of characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td><code>'Hello, World'</code>, <code>'Python\'s'</code></td>
</tr>
<tr>
<td>operations</td>
<td>concatenate, multiply</td>
</tr>
<tr>
<td>operators</td>
<td><code>+, *</code></td>
</tr>
</tbody>
</table>
ruler.py: The ruler function $R(n)$ is the exponent of the largest power of 2 which divides $2^n$. The $i$th row in the output lists the values of $R(n)$ for $n = 1, 2, \ldots, 2^i - 1$.

```python
import stdio

ruler1 = '1'
ruler2 = ruler1 + ' 2 ' + ruler1
ruler3 = ruler2 + ' 3 ' + ruler2
ruler4 = ruler3 + ' 4 ' + ruler3
stdio.writeln(ruler1)
stdio.writeln(ruler2)
stdio.writeln(ruler3)
stdio.writeln(ruler4)

$ python3 ruler.py
1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```
ruler.py: The ruler function $R(n)$ is the exponent of the largest power of 2 which divides $2^n$. The $i$th row in the output lists the values of $R(n)$ for $n = 1, 2, \ldots, 2^i - 1$.

```
import stdio

ruler1 = '1'
ruler2 = ruler1 + ' 2 ' + ruler1
ruler3 = ruler2 + ' 3 ' + ruler2
ruler4 = ruler3 + ' 4 ' + ruler3
stdio.writeln(ruler1)
stdio.writeln(ruler2)
stdio.writeln(ruler3)
stdio.writeln(ruler4)
```

$ python3 ruler.py
1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
ruler.py: The ruler function $R(n)$ is the exponent of the largest power of 2 which divides $2^n$. The $i$th row in the output lists the values of $R(n)$ for $n = 1, 2, \ldots, 2^i - 1$.

```python
import stdio

ruler1 = '1'
ruler2 = ruler1 + ' 2 ' + ruler1
ruler3 = ruler2 + ' 3 ' + ruler2
ruler4 = ruler3 + ' 4 ' + ruler3
stdio.writeln(ruler1)
stdio.writeln(ruler2)
stdio.writeln(ruler3)
stdio.writeln(ruler4)

$ python3 ruler.py
1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

9 / 23
Strings

The built-in function `str()` can be used to convert numbers into strings.

For example, `str(123)` evaluates to the `str` object `'123'`, and `str(123.45)` evaluates to the `str` object `'123.45'`.

The built-in functions `int()` and `float()` can be used to convert strings to numbers.

For example, `int('123')` is equivalent to the `int` literal `123`, and `float('123.45')` is equivalent to the `float` literal `123.45`. 
The built-in function `str()` can be used to convert numbers into strings

For example, `str(123)` evaluates to the `str` object `’123’`, and `str(123.45)` evaluates to the `str` object `’123.45’`

The built-in functions `int()` and `float()` can be used to convert strings to numbers

For example, `int(’123’)` is equivalent to the `int` literal `123`, and `float(’123.45’)` is equivalent to the `float` literal `123.45`
The built-in function `str()` can be used to convert numbers into strings

For example, `str(123)` evaluates to the `str` object `'123'`, and `str(123.45)` evaluates to the `str` object `'123.45'`

The built-in functions `int()` and `float()` can be used to convert strings to numbers

For example, `int('123')` is equivalent to the `int` literal 123, and `float('123.45')` is equivalent to the `float` literal 123.45
The built-in function `str()` can be used to convert numbers into strings

For example, `str(123)` evaluates to the `str` object `'123'`, and `str(123.45)` evaluates to the `str` object `'123.45'`

The built-in functions `int()` and `float()` can be used to convert strings to numbers

For example, `int('123')` is equivalent to the `int` literal 123, and `float('123.45')` is equivalent to the `float` literal 123.45
The built-in function `str()` can be used to convert numbers into strings

For example, `str(123)` evaluates to the `str` object `'123'`, and `str(123.45)` evaluates to the `str` object `'123.45'`

The built-in functions `int()` and `float()` can be used to convert strings to numbers

For example, `int('123')` is equivalent to the `int` literal 123, and `float('123.45')` is equivalent to the `float` literal 123.45
The *int* data type represents integers or natural numbers.

We can specify an *int* literal with a sequence of digits 0 through 9.

Python includes operators for common arithmetic operations on integers, including + for addition, - for subtraction, * for multiplication, // for floored division, % for remainder, and ** for exponentiation.

The *int* data type:

<table>
<thead>
<tr>
<th>values</th>
<th>integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>1234, 99, 0, 1000000</td>
</tr>
<tr>
<td>operations</td>
<td>sign, add, subtract, multiply, floored divide, remainder, power</td>
</tr>
<tr>
<td>operators</td>
<td>+, -, +, -, *, //, %, **</td>
</tr>
</tbody>
</table>
The `int` data type represents integers or natural numbers.

We can specify an `int` literal with a sequence of digits 0 through 9.

Python includes operators for common arithmetic operations on integers, including `+` for addition, `-` for subtraction, `*` for multiplication, `//` for floored division, `%` for remainder, and `**` for exponentiation.

The `int` data type:

<table>
<thead>
<tr>
<th>values</th>
<th>integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>1234, 99, 0, 1000000</td>
</tr>
<tr>
<td>operations</td>
<td>sign, add, subtract, multiply, floored divide, remainder, power</td>
</tr>
<tr>
<td>operators</td>
<td>+, -, *, //, %, **</td>
</tr>
</tbody>
</table>
The `int` data type represents integers or natural numbers

We can specify an `int` literal with a sequence of digits 0 through 9

Python includes operators for common arithmetic operations on integers, including + for addition, - for subtraction, * for multiplication, // for floored division, % for remainder, and ** for exponentiation

The `int` data type

<table>
<thead>
<tr>
<th>values</th>
<th>integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>1234, 99, 0, 1000000</td>
</tr>
<tr>
<td>operations</td>
<td>sign, add, subtract, multiply, floored divide, remainder, power</td>
</tr>
<tr>
<td>operators</td>
<td>+, -, +, -, *, //, %, **</td>
</tr>
</tbody>
</table>
The `int` data type represents integers or natural numbers.

We can specify an `int` literal with a sequence of digits 0 through 9.

Python includes operators for common arithmetic operations on integers, including + for addition, - for subtraction, * for multiplication, // for floored division, % for remainder, and ** for exponentiation.

<table>
<thead>
<tr>
<th>int data type values</th>
<th>integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>1234, 99, 0, 1000000</td>
</tr>
<tr>
<td>operations operators</td>
<td>sign, add, subtract, multiply, floored divide, remainder, power</td>
</tr>
<tr>
<td></td>
<td>+, -, *, //, %, **</td>
</tr>
</tbody>
</table>
The *int* data type represents integers or natural numbers.

We can specify an *int* literal with a sequence of digits 0 through 9.

Python includes operators for common arithmetic operations on integers, including + for addition, - for subtraction, * for multiplication, // for floored division, % for remainder, and ** for exponentiation.

<table>
<thead>
<tr>
<th>values</th>
<th>integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>1234, 99, 0, 1000000</td>
</tr>
<tr>
<td>operations</td>
<td>sign, add, subtract, multiply, floored divide, remainder, power</td>
</tr>
<tr>
<td>operators</td>
<td>+, -, *, //, %, **</td>
</tr>
</tbody>
</table>
Integers

intops.py: Accept two integers $a$ and $b$ as command-line arguments, perform integer operations on them, and write the results to standard output.

```python
import stdio
import sys

a = int(sys.argv[1])
b = int(sys.argv[2])
total = a + b
diff = a - b
prod = a * b
quot = a // b
rem = a % b
exp = a ** b
stdio.writeln(str(a) + ' + ' + str(b) + ' = ' + str(total))
stdio.writeln(str(a) + ' - ' + str(b) + ' = ' + str(diff))
stdio.writeln(str(a) + ' * ' + str(b) + ' = ' + str(prod))
stdio.writeln(str(a) + ' // ' + str(b) + ' = ' + str(quot))
stdio.writeln(str(a) + ' % ' + str(b) + ' = ' + str(rem))
stdio.writeln(str(a) + ' ** ' + str(b) + ' = ' + str(exp))
```

```
$ python3 intops.py 1234 5
1234 + 5 = 1239
1234 - 5 = 1229
1234 * 5 = 6170
1234 // 5 = 246
1234 % 5 = 4
1234 ** 5 = 2861381721051424
```
Integers

intops.py: Accept two integers $a$ and $b$ as command-line arguments, perform integer operations on them, and write the results to standard output.

```python
import sys

a = int(sys.argv[1])
b = int(sys.argv[2])
total = a + b
diff = a - b
prod = a * b
quot = a // b
rem = a % b
exp = a ** b

stdio.writeln(str(a) + ' + ' + str(b) + ' = ' + str(total))
stdio.writeln(str(a) + ' - ' + str(b) + ' = ' + str(diff))
stdio.writeln(str(a) + ' * ' + str(b) + ' = ' + str(prod))
stdio.writeln(str(a) + ' // ' + str(b) + ' = ' + str(quot))
stdio.writeln(str(a) + ' % ' + str(b) + ' = ' + str(rem))
stdio.writeln(str(a) + ' ** ' + str(b) + ' = ' + str(exp))
```

```
$ python3 intops.py 1234 5
1234 + 5 = 1239
1234 - 5 = 1229
1234 * 5 = 6170
1234 // 5 = 246
1234 % 5 = 4
1234 ** 5 = 2861381721051424
```
intops.py: Accept two integers $a$ and $b$ as command-line arguments, perform integer operations on them, and write the results to standard output.

```python
import stdio
import sys

a = int(sys.argv[1])
b = int(sys.argv[2])
total = a + b
diff = a - b
prod = a * b
quot = a // b
rem = a % b
exp = a ** b
stdio.writeln(str(a) + ' + ' + str(b) + ' = ' + str(total))
stdio.writeln(str(a) + ' - ' + str(b) + ' = ' + str(diff))
stdio.writeln(str(a) + ' * ' + str(b) + ' = ' + str(prod))
stdio.writeln(str(a) + ' // ' + str(b) + ' = ' + str(quot))
stdio.writeln(str(a) + ' % ' + str(b) + ' = ' + str(rem))
stdio.writeln(str(a) + ' ** ' + str(b) + ' = ' + str(exp))
```

```
$ python3 intops.py 1234 5
1234 + 5 = 1239
1234 - 5 = 1229
1234 * 5 = 6170
1234 // 5 = 246
1234 % 5 = 4
1234 ** 5 = 2861381721051424
```
Floating-point Numbers

The `float` data type represents floating-point numbers, for use in scientific and commercial applications.

We can specify a floating-point literal using a sequence of digits with a decimal point. For example, `3.14159` is a `float` literal that represents an approximation to π.

Alternatively, we can use a notation similar to scientific notation: the literal `6.022e23` represents the number `6.022 × 10^{23}`.

Python includes operators for common arithmetic operations on floating-point numbers, including `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `**` for exponentiation.

<table>
<thead>
<tr>
<th>values</th>
<th>real numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>3.14159, 6.022e23, 2.0, 1.4142135623730951</td>
</tr>
<tr>
<td>operations</td>
<td>sign, add, subtract, multiply, divide, power</td>
</tr>
<tr>
<td>operators</td>
<td>+, -, +, -, *, /, **</td>
</tr>
</tbody>
</table>
The `float` data type represents floating-point numbers, for use in scientific and commercial applications.

We can specify a floating-point literal using a sequence of digits with a decimal point. For example, `3.14159` is a `float` literal that represents an approximation to $\pi$.

Alternatively, we can use a notation similar to scientific notation: the literal `6.022e23` represents the number $6.022 \times 10^{23}$.

Python includes operators for common arithmetic operations on floating-point numbers, including `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `**` for exponentiation.

<table>
<thead>
<tr>
<th>values</th>
<th>real numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td><code>3.14159, 6.022e23, 2.0, 1.4142135623730951</code></td>
</tr>
<tr>
<td>operations</td>
<td>sign, add, subtract, multiply, divide, power</td>
</tr>
<tr>
<td>operators</td>
<td>+, -, +, -, *, /, **</td>
</tr>
</tbody>
</table>
Floating-point Numbers

The float data type represents floating-point numbers, for use in scientific and commercial applications.

We can specify a floating-point literal using a sequence of digits with a decimal point.

For example, 3.14159 is a float literal that represents an approximation to \( \pi \).

Alternatively, we can use a notation similar to scientific notation: the literal 6.022e23 represents the number \( 6.022 \times 10^{23} \).

Python includes operators for common arithmetic operations on floating-point numbers, including + for addition, - for subtraction, * for multiplication, / for division, and ** for exponentiation.

<table>
<thead>
<tr>
<th>values</th>
<th>real numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>3.14159, 6.022e23, 2.0, 1.4142135623730951</td>
</tr>
<tr>
<td>operations</td>
<td>sign, add, subtract, multiply, divide, power</td>
</tr>
<tr>
<td>operators</td>
<td>+, -, *, /, **</td>
</tr>
</tbody>
</table>
Floating-point Numbers

The float data type represents floating-point numbers, for use in scientific and commercial applications

We can specify a floating-point literal using a sequence of digits with a decimal point

For example, 3.14159 is a float literal that represents an approximation to $\pi$

Alternatively, we can use a notation similar to scientific notation: the literal 6.022e23 represents the number $6.022 \times 10^{23}$

Python includes operators for common arithmetic operations on floating-point numbers, including + for addition, - for subtraction, * for multiplication, / for division, and ** for exponentiation

The float data type

<table>
<thead>
<tr>
<th>values</th>
<th>real numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>3.14159, 6.022e23, 2.0, 1.4142135623730951</td>
</tr>
<tr>
<td>operations</td>
<td>sign, add, subtract, multiply, divide, power</td>
</tr>
<tr>
<td>operators</td>
<td>+, -, +, - , *, /, **</td>
</tr>
</tbody>
</table>
Floating-point Numbers

The float data type represents floating-point numbers, for use in scientific and commercial applications.

We can specify a floating-point literal using a sequence of digits with a decimal point.

For example, 3.14159 is a float literal that represents an approximation to π.

Alternatively, we can use a notation similar to scientific notation: the literal 6.022e23 represents the number $6.022 \times 10^{23}$.

Python includes operators for common arithmetic operations on floating-point numbers, including + for addition, - for subtraction, * for multiplication, / for division, and ** for exponentiation.

The float data type

<table>
<thead>
<tr>
<th>values</th>
<th>real numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>3.14159, 6.022e23, 2.0, 1.4142135623730951</td>
</tr>
<tr>
<td>operations</td>
<td>sign, add, subtract, multiply, divide, power</td>
</tr>
<tr>
<td>operators</td>
<td>+, -, +, -, *, /, **</td>
</tr>
</tbody>
</table>
Floating-point Numbers

The `float` data type represents floating-point numbers, for use in scientific and commercial applications.

We can specify a floating-point literal using a sequence of digits with a decimal point. For example, `3.14159` is a `float` literal that represents an approximation to $\pi$.

Alternatively, we can use a notation similar to scientific notation: the literal `6.022e23` represents the number $6.022 \times 10^{23}$.

Python includes operators for common arithmetic operations on floating-point numbers, including `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `**` for exponentiation.

<table>
<thead>
<tr>
<th>values</th>
<th>real numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>3.14159, 6.022e23, 2.0, 1.4142135623730951</td>
</tr>
<tr>
<td>operations</td>
<td>sign, add, subtract, multiply, divide, power</td>
</tr>
<tr>
<td>operators</td>
<td>+, -, +, -, *, /, **</td>
</tr>
</tbody>
</table>
Floating-point Numbers

The float data type represents floating-point numbers, for use in scientific and commercial applications.

We can specify a floating-point literal using a sequence of digits with a decimal point.

For example, $3.14159$ is a float literal that represents an approximation to $\pi$.

Alternatively, we can use a notation similar to scientific notation: the literal $6.022e23$ represents the number $6.022 \times 10^{23}$.

Python includes operators for common arithmetic operations on floating-point numbers, including + for addition, - for subtraction, * for multiplication, / for division, and ** for exponentiation.

The float data type

<table>
<thead>
<tr>
<th>values</th>
<th>real numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical literals</td>
<td>$3.14159$, $6.022e23$, $2.0$, $1.4142135623730951$</td>
</tr>
<tr>
<td>operations</td>
<td>sign, add, subtract, multiply, divide, power</td>
</tr>
<tr>
<td>operators</td>
<td>$+$, $-$, $\times$, $\div$, $^\ast$</td>
</tr>
</tbody>
</table>
Floating-point Numbers

floatops.py: Accept two floats \(a\) and \(b\) as command-line arguments, perform floating-point operations on them, and write the results to standard output.

```python
import stdio
import sys

a = float(sys.argv[1])
b = float(sys.argv[2])
total = a + b
diff = a - b
prod = a * b
quot = a / b
exp = a ** b
stdio.writeln(str(a) + ' + ' + str(b) + ' = ' + str(total))
stdio.writeln(str(a) + ' - ' + str(b) + ' = ' + str(diff))
stdio.writeln(str(a) + ' * ' + str(b) + ' = ' + str(prod))
stdio.writeln(str(a) + ' / ' + str(b) + ' = ' + str(quot))
stdio.writeln(str(a) + ' ** ' + str(b) + ' = ' + str(exp))
```

```
$ python3 floatops.py 123.456 78.9
123.456 + 78.9 = 202.356
123.456 - 78.9 = 44.556
123.456 * 78.9 = 9740.6784
123.456 / 78.9 = 1.5647148289
123.456 ** 78.9 = 1.04788279167e+165
```
floatops.py: Accept two floats $a$ and $b$ as command-line arguments, perform floating-point operations on them, and write the results to standard output.

```python
import stdio
import sys

a = float(sys.argv[1])
b = float(sys.argv[2])
total = a + b
diff = a - b
prod = a * b
quot = a / b
exp = a ** b
stdio.writeln(str(a) + ' + ' + str(b) + ' = ' + str(total))
stdio.writeln(str(a) + ' - ' + str(b) + ' = ' + str(diff))
stdio.writeln(str(a) + ' * ' + str(b) + ' = ' + str(prod))
stdio.writeln(str(a) + ' / ' + str(b) + ' = ' + str(quot))
stdio.writeln(str(a) + ' ** ' + str(b) + ' = ' + str(exp))
```

```
$ python3 floatops.py 123.456 78.9
123.456 + 78.9 = 202.356
123.456 - 78.9 = 44.556
123.456 * 78.9 = 9740.6784
123.456 / 78.9 = 1.5647148289
123.456 ** 78.9 = 1.04788279167e+165
```
Floating-point Numbers

floatops.py: Accept two floats \( a \) and \( b \) as command-line arguments, perform floating-point operations on them, and write the results to standard output.

```python
import stdio
import sys

a = float(sys.argv[1])
b = float(sys.argv[2])
total = a + b
diff = a - b
prod = a * b
quot = a / b
exp = a ** b
stdio.writeln(str(a) + ' + ' + str(b) + ' = ' + str(total))
stdio.writeln(str(a) + ' - ' + str(b) + ' = ' + str(diff))
stdio.writeln(str(a) + ' * ' + str(b) + ' = ' + str(prod))
stdio.writeln(str(a) + ' / ' + str(b) + ' = ' + str(quot))
stdio.writeln(str(a) + ' ** ' + str(b) + ' = ' + str(exp))
```

$ python3 floatops.py 123.456 78.9
123.456 + 78.9 = 202.356
123.456 - 78.9 = 44.556
123.456 * 78.9 = 9740.6784
123.456 / 78.9 = 1.5647148289
123.456 ** 78.9 = 1.04788279167e+165
Floating-point Numbers

quadratic.py: Accept floats \( b \) and \( c \) as command-line arguments, compute the roots of the polynomial \( x^2 + bx + c \) using the quadratic formula

\[
x = \frac{-b \pm \sqrt{b^2 - 4c}}{2},
\]

and write the roots to standard output.

```python
import math
import stdio
import sys

b = float(sys.argv[1])
c = float(sys.argv[2])
discriminant = b * b - 4.0 * c
d = math.sqrt(discriminant)
stdio.writeln((-b + d) / 2.0)
stdio.writeln((-b - d) / 2.0)
```

$ python3 quadratic.py -3.0 2.0
2.0
1.0

$ python3 quadratic.py -1.0 -1.0
1.61803398875
-0.61803398875

$ python quadratic.py 1.0 1.0
Traceback (most recent call last):
  File "quadratic.py", line 17, in <module>
    d = math.sqrt(discriminant)
ValueError: math domain error
Floating-point Numbers

quadratic.py: Accept floats $b$ and $c$ as command-line arguments, compute the roots of the polynomial $x^2 + bx + c$ using the quadratic formula $x = \frac{-b \pm \sqrt{b^2 - 4c}}{2}$, and write the roots to standard output.

```python
import math
import stdio
import sys

b = float(sys.argv[1])
c = float(sys.argv[2])
discriminant = b * b - 4.0 * c
d = math.sqrt(discriminant)
stdio.writeln((-b + d) / 2.0)
stdio.writeln((-b - d) / 2.0)
```

$ python3 quadratic.py -3.0 2.0
2.0
1.0
$ python3 quadratic.py -1.0 -1.0
1.61803398875
-0.61803398875
$ python quadratic.py 1.0 1.0
Traceback (most recent call last):
  File "quadratic.py", line 17, in <module>
    d = math.sqrt(discriminant)
ValueError: math domain error
Floating-point Numbers

quadratic.py: Accept floats $b$ and $c$ as command-line arguments, compute the roots of the polynomial $x^2 + bx + c$ using the quadratic formula $x = \frac{-b \pm \sqrt{b^2 - 4c}}{2}$, and write the roots to standard output.

```python
import math
import stdio
import sys

b = float(sys.argv[1])
c = float(sys.argv[2])
discriminant = b * b - 4.0 * c
d = math.sqrt(discriminant)
stdio.writeln((-b + d) / 2.0)
stdio.writeln((-b - d) / 2.0)
```

$ python3 quadratic.py -3.0 2.0
2.0
1.0

$ python3 quadratic.py -1.0 -1.0
1.61803398875
-0.61803398875

$ python quadratic.py 1.0 1.0
Traceback (most recent call last):
  File "quadratic.py", line 17, in <module>
    d = math.sqrt(discriminant)
ValueError: math domain error
The **bool** data type represents truth values (true or false) from logic.

The two **bool** literals are represented as **True** and **False**.

The operators defined for **bool** objects, namely **and**, **or**, and **not**, are known as logical operators, and having the following truth tables:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x and y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x or y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

The **bool** data type

- **values**: true, false
- **typical literals**: True, False
- **operations and operators**: and, or, not
The **bool** data type represents truth values (true or false) from logic.

The two **bool** literals are represented as **True** and **False**.

The operators defined for **bool** objects, namely **and**, **or**, and **not**, are known as logical operators, and having the following truth tables:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x and y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x or y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>not x</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
Booleans

The bool data type represents truth values (true or false) from logic

The two bool literals are represented as True and False

The operators defined for bool objects, namely and, or, and not, are known as logical operators, and having the following truth tables

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x and y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x or y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>not x</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
Booleans

The bool data type represents truth values (true or false) from logic.

The two bool literals are represented as True and False.

The operators defined for bool objects, namely and, or, and not, are known as logical operators, and having the following truth tables:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>x and y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>x or y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

The bool data type

- Values: true, false
- Typical literals: True, False
- Operations: and, or, not
- Operators: and, or, not
Booleans

The bool data type represents truth values (true or false) from logic

The two bool literals are represented as True and False

The operators defined for bool objects, namely and, or, and not, are known as logical operators, and having the following truth tables

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x and y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x or y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>not x</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
The comparison operators ==, !, <, <=, >, >=, is, and is not are defined for both integers and floats, and evaluate to a boolean result.

For example, $2 == 2$ evaluates to True, $2 == 3$ evaluates to False, $2 < 13$ evaluates to True.

Comparison operators have lower precedence than arithmetic operators and higher precedence than boolean operators, so you do not need the parentheses in an expression like $(b * b - 4.0 * a * c) >= 0.0$. 

The comparison operators `==`, `!=`, `<`, `<=`, `>`, `>=`, `is`, and `is not` are defined for both integers and floats, and evaluate to a boolean result.

For example, `2 == 2` evaluates to `True`, `2 == 3` evaluates to `False`, `2 < 13` evaluates to `True`.

Comparison operators have lower precedence than arithmetic operators and higher precedence than boolean operators, so you do not need the parentheses in an expression like `(b * b - 4.0 * a * c) >= 0.0`.
Booleans

The comparison operators ==, !=, <, <=, >, >=, is, and is not are defined for both integers and floats, and evaluate to a boolean result.

For example, 2 == 2 evaluates to True, 2 == 3 evaluates to False, 2 < 13 evaluates to True.

Comparison operators have lower precedence than arithmetic operators and higher precedence than boolean operators, so you do not need the parentheses in an expression like \((b \times b - 4.0 \times a \times c) \geq 0.0\).
Booleans

The comparison operators ==, !=, <, <=, >, >=, is, and is not are defined for both integers and floats, and evaluate to a boolean result.

For example, 2 == 2 evaluates to True, 2 == 3 evaluates to False, 2 < 13 evaluates to True.

Comparison operators have lower precedence than arithmetic operators and higher precedence than boolean operators, so you do not need the parentheses in an expression like $(b * b - 4.0 * a * c) >= 0.0$. 


leapyear.py: Accept an integer \( \text{\textit{year}} \) as command-line argument, and write \textbf{True} to standard output if \( \text{\textit{year}} \) is a leap year and \textbf{False} otherwise. A year is a leap year if it is divisible by 4 \textit{and} not divisible by 100 \textit{or} is divisible by 400.

```python
import stdio
import sys

year = int(sys.argv[1])
isLeapYear = (year % 4 == 0)
isLeapYear = isLeapYear and (year % 100 != 0)
isLeapYear = isLeapYear or (year % 400 == 0)
stdio.writeln(isLeapYear)
```

$ python3 leapyear.py 2016
True
$ python3 leapyear.py 1900
False
$ python3 leapyear.py 2000
True
Booleans

leapyear.py: Accept an integer \( year \) as command-line argument, and write True to standard output if \( year \) is a leap year and False otherwise. A year is a leap year if it is divisible by 4 and not divisible by 100 or is divisible by 400.

```python
import stdio
import sys

year = int(sys.argv[1])
isLeapYear = (year % 4 == 0)
isLeapYear = isLeapYear and (year % 100 != 0)
isLeapYear = isLeapYear or (year % 400 == 0)
stdio.writeln(isLeapYear)
```

$ python3 leapyear.py 2016
True
$ python3 leapyear.py 1900
False
$ python3 leapyear.py 2000
True
Booleans

leapyear.py: Accept an integer `year` as command-line argument, and write `True` to standard output if `year` is a leap year and `False` otherwise. A year is a leap year if it is divisible by 4 and not divisible by 100 or is divisible by 400.

```python
import stdio
import sys

year = int(sys.argv[1])
isLeapYear = (year % 4 == 0)
isLeapYear = isLeapYear and (year % 100 != 0)
isLeapYear = isLeapYear or (year % 400 == 0)
stdio.writeln(isLeapYear)
```

```
$ python3 leapyear.py 2016
True
$ python3 leapyear.py 1900
False
$ python3 leapyear.py 2000
True
```
Functions and APIs

Many programming tasks involve not only built-in operators, but also functions.

We consider three kinds of functions:

1. **Built-in functions** (such as `int()`, `float()`, and `str()`) that you can use directly in any Python program.
2. **Standard functions** (such as `math.sqrt()`) that are defined in a Python standard module and are available in any program that imports the module.
3. **User-defined functions** (such as `stdio.write()` and `stdio.writeln()`) that are defined in third-party modules.

We can call a function in our code by typing its name followed by arguments (which are just expressions), enclosed in parentheses and separated by commas.

For example, `math.sqrt(2.0)` is a function call.

When Python executes your program, we say that it calls (or evaluates) the function with the given arguments.

A function call that returns a value is an expression, so we can use it in the same way that we use variables and literals to build up more complicated expressions.

For example, `math.sin(x) * math.cos(y)` is an expression.

A function call that does not return a value, but has a side effect, can only be used as a statement.

For example, `stdio.writeln('Hello, World')` is a statement.
Functions and APIs

Many programming tasks involve not only built-in operators, but also functions

We consider three kinds of functions

1. Built-in functions (such as \texttt{int()}, \texttt{float()}, and \texttt{str()}) that you can use directly in any Python program
2. Standard functions (such as \texttt{math.sqrt()}) that are defined in a Python standard module and are available in any program that imports the module
3. User-defined functions (such as \texttt{stdio.write()} and \texttt{stdio.writeln()}) that are defined in third-party modules

We can call a function in our code by typing its name followed by arguments (which are just expressions), enclosed in parentheses and separated by commas.

For example, \texttt{math.sqrt(2.0)} is a function call.

When Python executes your program, we say that it calls (or evaluates) the function with the given arguments.

A function call that returns a value is an expression, so we can use it in the same way that we use variables and literals to build up more complicated expressions.

For example, \texttt{math.sin(x) \ast math.cos(y)} is an expression.

A function call that does not return a value, but has a side effect, can only be used as a statement.

For example, \texttt{stdio.writeln('Hello, World')} is a statement.
Functions and APIs

Many programming tasks involve not only built-in operators, but also functions

We consider three kinds of functions

1. **Built-in functions** (such as `int()`, `float()`, and `str()`) that you can use directly in any Python program
2. **Standard functions** (such as `math.sqrt()`) that are defined in a Python standard module and are available in any program that imports the module
3. **User-defined functions** (such as `stdio.write()` and `stdio.writeln()`) that are defined in third-party modules

We can call a function in our code by typing its name followed by arguments (which are just expressions), enclosed in parentheses and separated by commas.

For example, `math.sqrt(2.0)` is a function call.

When Python executes your program, we say that it calls (or evaluates) the function with the given arguments.

A function call that returns a value is an expression, so we can use it in the same way that we use variables and literals to build up more complicated expressions.

For example, `math.sin(x) * math.cos(y)` is an expression.

A function call that does not return a value, but has a side effect, can only be used as a statement.

For example, `stdio.writeln('Hello, World')` is a statement.
Functions and APIs

Many programming tasks involve not only built-in operators, but also functions

We consider three kinds of functions

1. Built-in functions (such as int(), float(), and str()) that you can use directly in any Python program
2. Standard functions (such as math.sqrt()) that are defined in a Python standard module and are available in any program that imports the module
3. User-defined functions (such as stdio.write() and stdio.writeln()) that are defined in third-party modules

We can call a function in our code by typing its name followed by arguments (which are just expressions), enclosed in parentheses and separated by commas

For example, math.sqrt(2.0) is a function call

When Python executes your program, we say that it calls (or evaluates) the function with the given arguments.

A function call that returns a value is an expression, so we can use it in the same way that we use variables and literals to build up more complicated expressions

For example, math.sin(x) * math.cos(y) is an expression

A function call that does not return a value, but has a side effect, can only be used as a statement

For example, stdio.writeln('Hello, World') is a statement
Functions and APIs

Many programming tasks involve not only built-in operators, but also functions

We consider three kinds of functions

1. **Built-in functions** (such as `int()`, `float()`, and `str()`) that you can use directly in any Python program
2. **Standard functions** (such as `math.sqrt()`) that are defined in a Python standard module and are available in any program that imports the module
3. **User-defined functions** (such as `stdio.write()` and `stdio.writeln()`) that are defined in third-party modules

We can call a function in our code by typing its name followed by arguments (which are just expressions), enclosed in parentheses and separated by commas

For example, `math.sqrt(2.0)` is a function call

When Python executes your program, we say that it calls (or evaluates) the function with the given arguments.

A function call that returns a value is an expression, so we can use it in the same way that we use variables and literals to build up more complicated expressions

For example, `math.sin(x) * math.cos(y)` is an expression

A function call that does not return a value, but has a side effect, can only be used as a statement

For example, `stdio.writeln('Hello, World')` is a statement
Functions and APIs

Many programming tasks involve not only built-in operators, but also functions.

We consider three kinds of functions:

1. Built-in functions (such as `int()`, `float()`, and `str()`) that you can use directly in any Python program.
2. Standard functions (such as `math.sqrt()`) that are defined in a Python standard module and are available in any program that imports the module.
3. User-defined functions (such as `stdio.write()` and `stdio.writeln()`) that are defined in third-party modules.

We can call a function in our code by typing its name followed by arguments (which are just expressions), enclosed in parentheses and separated by commas.

For example, `math.sqrt(2.0)` is a function call.

When Python executes your program, we say that it calls (or evaluates) the function with the given arguments.

A function call that returns a value is an expression, so we can use it in the same way that we use variables and literals to build up more complicated expressions.

For example, `math.sin(x) * math.cos(y)` is an expression.

A function call that does not return a value, but has a side effect, can only be used as a statement.

For example, `stdio.writeln('Hello, World')` is a statement.
Functions and APIs

Many programming tasks involve not only built-in operators, but also functions.

We consider three kinds of functions:

1. Built-in functions (such as `int()`, `float()`, and `str()`) that you can use directly in any Python program.
2. Standard functions (such as `math.sqrt()`) that are defined in a Python standard module and are available in any program that imports the module.
3. User-defined functions (such as `stdio.write()` and `stdio.writeln()`) that are defined in third-party modules.

We can call a function in our code by typing its name followed by arguments (which are just expressions), enclosed in parentheses and separated by commas.

For example, `math.sqrt(2.0)` is a function call.

When Python executes your program, we say that it calls (or evaluates) the function with the given arguments.

A function call that returns a value is an expression, so we can use it in the same way that we use variables and literals to build up more complicated expressions.

For example, `math.sin(x) * math.cos(y)` is an expression.

A function call that does not return a value, but has a side effect, can only be used as a statement.

For example, `stdio.writeln('Hello, World')` is a statement.
Functions and APIs

Many programming tasks involve not only built-in operators, but also functions.

We consider three kinds of functions:

1. Built-in functions (such as `int()`, `float()`, and `str()`) that you can use directly in any Python program.
2. Standard functions (such as `math.sqrt()`) that are defined in a Python standard module and are available in any program that imports the module.
3. User-defined functions (such as `stdio.write()` and `stdio.writeln()`) that are defined in third-party modules.

We can call a function in our code by typing its name followed by arguments (which are just expressions), enclosed in parentheses and separated by commas.

For example, `math.sqrt(2.0)` is a function call.

When Python executes your program, we say that it calls (or evaluates) the function with the given arguments.

A function call that returns a value is an expression, so we can use it in the same way that we use variables and literals to build up more complicated expressions.

For example, `math.sin(x) * math.cos(y)` is an expression.

A function call that does not return a value, but has a side effect, can only be used as a statement.

For example, `stdio.writeln('Hello, World')` is a statement.
Functions and APIs

Many programming tasks involve not only built-in operators, but also functions

We consider three kinds of functions

1. Built-in functions (such as `int()`, `float()`, and `str()`) that you can use directly in any Python program
2. Standard functions (such as `math.sqrt()`) that are defined in a Python standard module and are available in any program that imports the module
3. User-defined functions (such as `stdio.write()` and `stdio.writeln()`) that are defined in third-party modules

We can call a function in our code by typing its name followed by arguments (which are just expressions), enclosed in parentheses and separated by commas.

For example, `math.sqrt(2.0)` is a function call

When Python executes your program, we say that it calls (or evaluates) the function with the given arguments.

A function call that returns a value is an expression, so we can use it in the same way that we use variables and literals to build up more complicated expressions.

For example, `math.sin(x) * math.cos(y)` is an expression

A function call that does not return a value, but has a side effect, can only be used as a statement.

For example, `stdio.writeln('Hello, World')` is a statement
Functions and APIs

Many programming tasks involve not only built-in operators, but also functions.

We consider three kinds of functions:

1. Built-in functions (such as `int()`, `float()`, and `str()`) that you can use directly in any Python program
2. Standard functions (such as `math.sqrt()`) that are defined in a Python standard module and are available in any program that imports the module
3. User-defined functions (such as `stdio.write()` and `stdio.writeln()`) that are defined in third-party modules

We can call a function in our code by typing its name followed by arguments (which are just expressions), enclosed in parentheses and separated by commas.

For example, `math.sqrt(2.0)` is a function call.

When Python executes your program, we say that it calls (or evaluates) the function with the given arguments.

A function call that returns a value is an expression, so we can use it in the same way that we use variables and literals to build up more complicated expressions.

For example, `math.sin(x) * math.cos(y)` is an expression.

A function call that does not return a value, but has a side effect, can only be used as a statement.

For example, `stdio.writeln('Hello, World')` is a statement.
**Functions and APIs**

We summarize functions in a table called the application programming interface (API).

**Built-in functions**

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>absolute value of x</td>
</tr>
<tr>
<td>max(a, b)</td>
<td>maximum value of a and b</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Standard functions from Python’s math and random modules**

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>math.sin(x)</td>
<td>sine of x (expressed in radians)</td>
</tr>
<tr>
<td>math.cos(x)</td>
<td>cosine of x (expressed in radians)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random.random()</td>
<td>a random float from the real interval [0, 1)</td>
</tr>
<tr>
<td>random.randrange(x, y)</td>
<td>a random integer from the integer interval [x, y)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**User-defined functions from thestdio module**

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdio.write(x)</td>
<td>write x to standard output</td>
</tr>
<tr>
<td>stdio.writeln(x)</td>
<td>write x to standard output, followed by a newline</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Functions and APIs

We summarize functions in a table called the application programming interface (API)

Built-in functions

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>absolute value of x</td>
</tr>
<tr>
<td>max(a, b)</td>
<td>maximum value of a and b</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Standard functions from Python’s `math` and `random` modules

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>math.sin(x)</td>
<td>sine of x (expressed in radians)</td>
</tr>
<tr>
<td>math.cos(x)</td>
<td>cosine of x (expressed in radians)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random.random()</td>
<td>a random float from the real interval [0, 1)</td>
</tr>
<tr>
<td>random.randrange(x, y)</td>
<td>a random integer from the integer interval [x, y)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

User-defined functions from the `stdio` module

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdio.write(x)</td>
<td>write x to standard output</td>
</tr>
<tr>
<td>stdio.writeln(x)</td>
<td>write x to standard output, followed by a newline</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
## Functions and APIs

We summarize functions in a table called the application programming interface (API)

### Built-in functions

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>absolute value of $x$</td>
</tr>
<tr>
<td>max(a, b)</td>
<td>maximum value of $a$ and $b$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Standard functions from Python’s `math` and `random` modules

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>math.sin(x)</td>
<td>sine of $x$ (expressed in radians)</td>
</tr>
<tr>
<td>math.cos(x)</td>
<td>cosine of $x$ (expressed in radians)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random.random()</td>
<td>a random float from the real interval [0, 1)</td>
</tr>
<tr>
<td>random.randrange(x, y)</td>
<td>a random integer from the integer interval $[x, y)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### User-defined functions from the `stdio` module

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdio.write(x)</td>
<td>write $x$ to standard output</td>
</tr>
<tr>
<td>stdio.writeln(x)</td>
<td>write $x$ to standard output, followed by a newline</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
**Functions and APIs**

We summarize functions in a table called the application programming interface (API)

**Built-in functions**

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs(x)</code></td>
<td>absolute value of <code>x</code></td>
</tr>
<tr>
<td><code>max(a, b)</code></td>
<td>maximum value of <code>a</code> and <code>b</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Standard functions from Python’s **math** and **random** modules**

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>math.sin(x)</code></td>
<td>sine of <code>x</code> (expressed in radians)</td>
</tr>
<tr>
<td><code>math.cos(x)</code></td>
<td>cosine of <code>x</code> (expressed in radians)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>random.random()</code></td>
<td>a random float from the real interval <code>[0, 1)</code></td>
</tr>
<tr>
<td><code>random.randrange(x, y)</code></td>
<td>a random integer from the integer interval <code>[x, y)</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**User-defined functions from the **stdio** module**

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stdio.write(x)</code></td>
<td>write <code>x</code> to standard output</td>
</tr>
<tr>
<td><code>stdio.writeln(x)</code></td>
<td>write <code>x</code> to standard output, followed by a newline</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Functions and APIs

We summarize functions in a table called the application programming interface (API)

Built-in functions

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>absolute value of x</td>
</tr>
<tr>
<td>max(a, b)</td>
<td>maximum value of a and b</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Standard functions from Python’s math and random modules

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>math.sin(x)</td>
<td>sine of x (expressed in radians)</td>
</tr>
<tr>
<td>math.cos(x)</td>
<td>cosine of x (expressed in radians)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random.random()</td>
<td>a random float from the real interval [0, 1)</td>
</tr>
<tr>
<td>random.randrange(x, y)</td>
<td>a random integer from the integer interval [x, y)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

User-defined functions from the stdio module

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdio.write(x)</td>
<td>write x to standard output</td>
</tr>
<tr>
<td>stdio.writeln(x)</td>
<td>write x to standard output, followed by a newline</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Type Conversion

We can use built-in functions \texttt{int()}, \texttt{float()}, \texttt{str()}, and \texttt{round()} to explicitly convert from strings to integers or floats, and vice versa.

Python also supports implicit conversion (aka automatic promotion or coercion).

For example, we can use an integer where a float is expected, as in \texttt{math.sqrt(4)}, which evaluates to \texttt{2.0}.
We can use built-in functions `int()`, `float()`, `str()`, and `round()` to explicitly convert from strings to integers or floats, and vice versa.

Python also supports implicit conversion (aka automatic promotion or coercion).

For example, we can use an integer where a float is expected, as in `math.sqrt(4)`, which evaluates to `2.0`.
Type Conversion

We can use built-in functions `int()`, `float()`, `str()`, and `round()` to explicitly convert from strings to integers or floats, and vice versa.

Python also supports implicit conversion (aka automatic promotion or coercion).

For example, we can use an integer where a float is expected, as in `math.sqrt(4)`, which evaluates to `2.0`.
Type Conversion

We can use built-in functions `int()`, `float()`, `str()`, and `round()` to explicitly convert from strings to integers or floats, and vice versa.

Python also supports implicit conversion (aka automatic promotion or coercion).

For example, we can use an integer where a float is expected, as in `math.sqrt(4)`, which evaluates to 2.0.
Interactive Python

We can use Python as a calculator by the running command python3 in the terminal

$ python3
...
>>> 1 + 2
3
>>> a = 1
>>> b = 2
>>> a + b
3
>>> import math
>>> math.sqrt(2.0)
1.4142135623730951
>>> math.e
2.718281828459045

We can type dir() without arguments to get a list of names in the current local scope; With an object argument, we get a list of valid attributes for that object

>>> dir()
[ '__builtins__', '__doc__', '__name__', '__package__', 'a', 'b', 'math']
>>> dir(math)

22 / 23
We can use Python as a calculator by the running command `python3` in the terminal

```
$ python3
...
>>> 1 + 2
3
>>> a = 1
>>> b = 2
>>> a + b
3
>>> import math
>>> math.sqrt(2.0)
1.4142135623730951
>>> math.e
2.718281828459045
```

We can type `dir()` without arguments to get a list of names in the current local scope; With an object argument, we get a list of valid attributes for that object

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'b', 'math']
>>> dir(math)
```
We can use Python as a calculator by the running command `python3` in the terminal.

```
$ python3
...
>>> 1 + 2
3
>>> a = 1
>>> b = 2
>>> a + b
3
>>> import math
>>> math.sqrt(2.0)
1.4142135623730951
>>> math.e
2.718281828459045
```

We can type `dir()` without arguments to get a list of names in the current local scope; With an object argument, we get a list of valid attributes for that object.

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'b', 'math']
>>> dir(math)
```
Interactive Python

We can type `help()` to get access to Python’s extensive interactive documentation

```python
>>> help(math)
Help on built-in module math:

NAME
    math

FILE
    (built-in)

DESCRIPTION
    This module is always available. It provides access to the mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

... DATA
    e = 2.718281828459045
    pi = 3.141592653589793
```

We can type `exit()` to return to the terminal

```python
>>> exit()
$```
Interactive Python

We can type `help()` to get access to Python’s extensive interactive documentation

```python
>>> help(math)
Help on built-in module math:

NAME
    math

FILE
    (built-in)

DESCRIPTION
    This module is always available. It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)  
        acos(x)
        
            Return the arc cosine (measured in radians) of x.

...  

DATA
    e = 2.718281828459045
    pi = 3.141592653589793
```

We can type `exit()` to return to the terminal

```python
>>> exit()
```

$
Interactive Python

We can type `help()` to get access to Python’s extensive interactive documentation

```python
>>> help(math)
Help on built-in module math:

NAME
    math

FILE
    (built-in)

DESCRIPTION
    This module is always available. It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

...  

DATA
    e = 2.718281828459045
    pi = 3.141592653589793
```

We can type `exit()` to return to the terminal

```python
>>> exit()
$  
```