Basic Data Types

Outline

Types

Types

A data type is a set of values along with a set of operations defined on those values

Types

A data type is a set of values along with a set of operations defined on those values

The four basic data types:

Types

A data type is a set of values along with a set of operations defined on those values

The four basic data types:

1. `str` for sequences of characters

Types

A data type is a set of values along with a set of operations defined on those values

The four basic data types:

1. `str` for sequences of characters
2. `int` for integers

Types

A data type is a set of values along with a set of operations defined on those values

The four basic data types:

1. `str` for sequences of characters
2. `int` for integers
3. `float` for floating-point numbers

Types

A data type is a set of values along with a set of operations defined on those values

The four basic data types:

1. `str` for sequences of characters
2. `int` for integers
3. `float` for floating-point numbers
4. `bool` for true/false values

Expressions

Expressions

A literal is a representation of a data-type value

Expressions

A literal is a representation of a data-type value

Example:

Expressions

A literal is a representation of a data-type value

Example:

- `'Hello, World'` and `'Cogito, ergo sum'` are string literals

Expressions

A literal is a representation of a data-type value

Example:

- `'Hello, World'` and `'Cogito, ergo sum'` are string literals
- `42` and `1729` are integer literals

Expressions

A literal is a representation of a data-type value

Example:

- `'Hello, World'` and `'Cogito, ergo sum'` are string literals
- `42` and `1729` are integer literals
- `3.14159` and `2.71828` are floating-point literals

Expressions

A literal is a representation of a data-type value

Example:

- `'Hello, World'` and `'Cogito, ergo sum'` are string literals
- `42` and `1729` are integer literals
- `3.14159` and `2.71828` are floating-point literals
- `True` and `False` are boolean literals

Expressions

Expressions

An identifier is a representation of a name

Expressions

An identifier is a representation of a name

Each identifier is a sequence of letters, digits, and underscores, not starting with a digit

Expressions

An identifier is a representation of a name

Each identifier is a sequence of letters, digits, and underscores, not starting with a digit

Example:

Expressions

An identifier is a representation of a name

Each identifier is a sequence of letters, digits, and underscores, not starting with a digit

Example:

- `abc`, `Ab_`, `abc123`, and `a_b` are valid identifiers

Expressions

An identifier is a representation of a name

Each identifier is a sequence of letters, digits, and underscores, not starting with a digit

Example:

- `abc`, `Ab_`, `abc123`, and `a_b` are valid identifiers
- `Ab*`, `1abc`, and `a+b` are not

Expressions

An identifier is a representation of a name

Each identifier is a sequence of letters, digits, and underscores, not starting with a digit

Example:

- `abc`, `Ab_`, `abc123`, and `a_b` are valid identifiers
- `Ab*`, `1abc`, and `a+b` are not

Keywords such as `and`, `def`, `import`, `lambda`, and `while` cannot be used as identifiers

Expressions

Expressions

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

Expressions

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

Example: `total` representing the running total of a sequence of numbers

Expressions

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

Example: `total` representing the running total of a sequence of numbers

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

Expressions

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

Example: `total` representing the running total of a sequence of numbers

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

Example: `SPEED_OF_LIGHT` representing the known speed of light

Expressions

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

Example: `total` representing the running total of a sequence of numbers

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

Example: `SPEED_OF_LIGHT` representing the known speed of light

A variable's value is accessed as `[<target>.]<name>`

Expressions

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

Example: `total` representing the running total of a sequence of numbers

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

Example: `SPEED_OF_LIGHT` representing the known speed of light

A variable's value is accessed as `[<target>.]<name>`

Example: `total`, `SPEED_OF_LIGHT`, `sys.argv`, and `math.pi`

Expressions

Expressions

An operator is a representation of a data-type operation

Expressions

An operator is a representation of a data-type operation

`+`, `-`, `*`, `/`, and `%` represent arithmetic operations on integers and floats

Expressions

An operator is a representation of a data-type operation

`+`, `-`, `*`, `/`, and `%` represent arithmetic operations on integers and floats

`not`, `or`, and `and` represent logical operations on booleans

Expressions

Expressions

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

Expressions

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

Three kinds of functions:

Expressions

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

Three kinds of functions:

1. Built-in functions

Expressions

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

Three kinds of functions:

1. Built-in functions
2. Functions defined in standard libraries

Expressions

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

Three kinds of functions:

1. Built-in functions
2. Functions defined in standard libraries
3. Functions defined in user-defined libraries

Expressions

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

Three kinds of functions:

1. Built-in functions
2. Functions defined in standard libraries
3. Functions defined in user-defined libraries

A function is called as `[<library>.]<name>(<argument1>, <argument2>, ...)`

Expressions

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

Three kinds of functions:

1. Built-in functions
2. Functions defined in standard libraries
3. Functions defined in user-defined libraries

A function is called as `[<library>.]<name>(<argument1>, <argument2>, ...)`

Example: `stdio.writeln('Hello, World')`

Expressions

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

Three kinds of functions:

1. Built-in functions
2. Functions defined in standard libraries
3. Functions defined in user-defined libraries

A function is called as `[<library>.]<name>(<argument1>, <argument2>, ...)`

Example: `stdio.writeln('Hello, World')`

Some functions (called void functions) do not return a value while others (called non-void functions) do return a value

Expressions

Expressions

Example

Expressions

Example

`int(x)`	returns the integer value of `x`
`float(x)`	returns the floating-point value of `x`
`str(x)`	returns string value of `x`

Example

`int(x)`	returns the integer value of `x`
`float(x)`	returns the floating-point value of `x`
`str(x)`	returns string value of `x`

`math`

`exp(x)`	returns e^{x}
`sqrt(x)`	returns $\sqrt{x}$

Expressions

Example

`int(x)`	returns the integer value of `x`
`float(x)`	returns the floating-point value of `x`
`str(x)`	returns string value of `x`

math

`exp(x)`	returns e^{x}
`sqrt(x)`	returns $\sqrt{x}$

stdio

`writeln(x = '')`	writes `x` followed by newline to standard output
`write(x = '')`	writes `x` to standard output

Example

`int(x)`	returns the integer value of `x`
`float(x)`	returns the floating-point value of `x`
`str(x)`	returns string value of `x`

math

`exp(x)`	returns e^x
`sqrt(x)`	returns $\sqrt{x}$

stdio

`writeln(x = '')`	writes `x` followed by newline to standard output
`write(x = '')`	writes `x` to standard output

stdrandom

`uniformFloat(lo, hi)`	returns a float chosen uniformly at random from the interval `[lo, hi)`
`bernoulli(p = 0.5)`	returns `True` with probability `p` and `False` with probability `1 - p`

Expressions

Expressions

An expression is a combination of literals, variables, operators, and non-void function calls that evaluates to a value

Expressions

An expression is a combination of literals, variables, operators, and non-void function calls that evaluates to a value

Example:

Expressions

An expression is a combination of literals, variables, operators, and non-void function calls that evaluates to a value

Example:

- `2, 4`

Expressions

An expression is a combination of literals, variables, operators, and non-void function calls that evaluates to a value

Example:

- `2, 4`
- `a, b, c`

Expressions

An expression is a combination of literals, variables, operators, and non-void function calls that evaluates to a value

Example:

- `2, 4`
- `a, b, c`
- `b * b - 4 * a * c`

Expressions

An expression is a combination of literals, variables, operators, and non-void function calls that evaluates to a value

Example:

- `2, 4`
- `a, b, c`
- `b * b - 4 * a * c`
- `math.sqrt(b * b - 4 * a * c)`

Expressions

An expression is a combination of literals, variables, operators, and non-void function calls that evaluates to a value

Example:

- `2, 4`
- `a, b, c`
- `b * b - 4 * a * c`
- `math.sqrt(b * b - 4 * a * c)`
- `(-b + math.sqrt(b * b - 4 * a * c)) / (2 * a)`

Statements

Statements

A statement is a syntactic unit that expresses some action to be carried out

Statements

A statement is a syntactic unit that expresses some action to be carried out

Import statement

```
import <library>
```

Statements

A statement is a syntactic unit that expresses some action to be carried out

Import statement

```
import <library>
```

Example

```
import stdio
import sys
```

Statements

Statements

Function call statement

```
[<library>.]<name>(<argument1>, <argument2>, ...)
```

Statements

Function call statement

```
[<library>.]<name>(<argument1>, <argument2>, ...)
```

Example

```
stdio.write('Cogito, ')
stdio.write('ergo sum')
stdio.writeln()
```

Statements

Statements

Assignment statement

```
<name> = <expression>
```

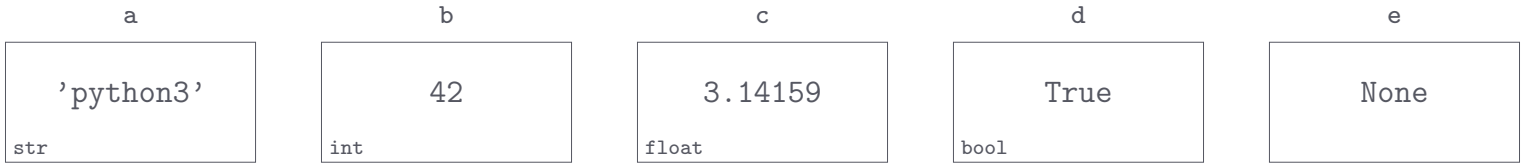
Statements

Assignment statement

```
<name> = <expression>
```

Example

```
a = 'python3'
b = 42
c = 3.14159
d = True
e = None
```



Statements

Statements

Example (exchanging the values of two variables `a` and `b`)

```
a = 42
b = 1729

t = a # t is now 42
a = b # a is now 1729
b = t # b is now 42

stdio.writeln(a)
stdio.writeln(b)
```

Statements

Example (exchanging the values of two variables `a` and `b`)

```
a = 42
b = 1729

t = a # t is now 42
a = b # a is now 1729
b = t # b is now 42

stdio.writeln(a)
stdio.writeln(b)
```

```
1729
42
```

Statements

Statements

Equivalent assignment statement forms

```
<name> <operator>= <expression>
<name> = <name> <operator> <expression>
```

where `<operator>` is `**`, `*`, `/`, `//`, `%`, `+`, or `-`

Statements

Equivalent assignment statement forms

```
<name> <operator>= <expression>
<name> = <name> <operator> <expression>
```

where `<operator>` is `**`, `*`, `/`, `//`, `%`, `+`, or `-`

Example

```
x *= 5
x = x * 5
```

Strings

Strings

The `str` data type represents strings (sequences of characters)

Strings

The `str` data type represents strings (sequences of characters)

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

Strings

The `str` data type represents strings (sequences of characters)

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

Example: `'Hello, World'` and `'Cogito, ergo sum'`

Strings

The `str` data type represents strings (sequences of characters)

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

Example: `'Hello, World'` and `'Cogito, ergo sum'`

Tab, newline, backslash, and single quote characters are specified using escape sequences `'\t'`, `'\n'`, `'\\'`, and `'\''`

Strings

The `str` data type represents strings (sequences of characters)

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

Example: `'Hello, World'` and `'Cogito, ergo sum'`

Tab, newline, backslash, and single quote characters are specified using escape sequences `'\t'`, `'\n'`, `'\\'`, and `'\''`

Example: `'Hello, world\n'` and `'Python\'s great'`

Strings

The `str` data type represents strings (sequences of characters)

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

Example: `'Hello, World'` and `'Cogito, ergo sum'`

Tab, newline, backslash, and single quote characters are specified using escape sequences `'\t'`, `'\n'`, `'\\'`, and `'\''`

Example: `'Hello, world\n'` and `'Python\'s great'`

Operations:

Strings

The `str` data type represents strings (sequences of characters)

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

Example: `'Hello, World'` and `'Cogito, ergo sum'`

Tab, newline, backslash, and single quote characters are specified using escape sequences `'\t'`, `'\n'`, `'\\'`, and `'\''`

Example: `'Hello, world\n'` and `'Python\'s great'`

Operations:

- Concatenation (`+`)

  Example: `'123' + '456'` evaluates to `'123456'`

Strings

The `str` data type represents strings (sequences of characters)

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

Example: `'Hello, World'` and `'Cogito, ergo sum'`

Tab, newline, backslash, and single quote characters are specified using escape sequences `'\t'`, `'\n'`, `'\\'`, and `'\''`

Example: `'Hello, world\n'` and `'Python\'s great'`

Operations:

- Concatenation (`+`)

  Example: `'123' + '456'` evaluates to `'123456'`
- Replication (`*`)

  Example: `3 * 'ab'` and `'ab' * 3` evaluate to `'ababab'`

Strings

Strings

Program: `dateformats.py`

Program: `dateformats.py`

- Command-line input: d (str), m (str), and y (str) representing a date

Program: `dateformats.py`

- Command-line input: d (str), m (str), and y (str) representing a date
- Standard output: the date in different formats

Strings

Program: `dateformats.py`

- Command-line input: *d* (str), *m* (str), and *y* (str) representing a date
- Standard output: the date in different formats

```
>_ ~/workspace/ipp/programs
$ _
```

Strings

Program: `dateformats.py`

- Command-line input: *d* (str), *m* (str), and *y* (str) representing a date
- Standard output: the date in different formats

```
>_ ~/workspace/ipp/programs
$ python3 dateformats.py 14 03 1879
```

Strings

Program: `dateformats.py`

- Command-line input: *d* (str), *m* (str), and *y* (str) representing a date
- Standard output: the date in different formats

```
>_ ~/workspace/ipp/programs
```

```
$ python3 dateformats.py 14 03 1879
14/03/1879
03/14/1879
1879/03/14
$ _
```

Strings

Strings

dateformats.py

```
1  import stdio
2  import sys
3
4  d = sys.argv[1]
5  m = sys.argv[2]
6  y = sys.argv[3]
7  dmy = d + '/' + m + '/' + y
8  mdy = m + '/' + d + '/' + y
9  ymd = y + '/' + m + '/' + d
10 stdio.writeln(dmy)
11 stdio.writeln(mdy)
12 stdio.writeln(ymd)
```

Integers

Integers

The `int` data type represents integers

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Example: `42` and `1729`

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Example: `42` and `1729`

Operations:

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Example: `42` and `1729`

Operations:

- Addition (`+`)

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Example: `42` and `1729`

Operations:

- Addition (`+`)
- Subtraction/negation (`-`)

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Example: `42` and `1729`

Operations:

- Addition (`+`)
- Subtraction/negation (`-`)
- Multiplication (`*`)

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Example: `42` and `1729`

Operations:

- Addition (`+`)
- Subtraction/negation (`-`)
- Multiplication (`*`)
- Division (`/`)

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Example: `42` and `1729`

Operations:

- Addition (`+`)
- Subtraction/negation (`-`)
- Multiplication (`*`)
- Division (`/`)
- Floored division(`//`)

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Example: `42` and `1729`

Operations:

- Addition (`+`)
- Subtraction/negation (`-`)
- Multiplication (`*`)
- Division (`/`)
- Floored division(`//`)
- Remainder (`%`)

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Example: `42` and `1729`

Operations:

- Addition (`+`)
- Subtraction/negation (`-`)
- Multiplication (`*`)
- Division (`/`)
- Floored division(`//`)
- Remainder (`%`)
- Exponentiation (`**`)

Integers

Integers

Program: `sumofsquares.py`

Integers

Program: `sumofsquares.py`

- Command-line input: x (int) and y (int)

Integers

Program: `sumofsquares.py`

- Command-line input: x (int) and y (int)
- Standard output: $x^2 + y^2$

Integers

Program: `sumofsquares.py`

- Command-line input: x (int) and y (int)
- Standard output: $x^2 + y^2$

```
>_ ~/workspace/ipp/programs
$ _
```

Integers

Program: `sumofsquares.py`

- Command-line input: x (int) and y (int)
- Standard output: $x^2 + y^2$

```
>_ ~/workspace/ipp/programs
$ python3 sumofsquares.py 3 4
```

Integers

Program: `sumofsquares.py`

- Command-line input: x (int) and y (int)
- Standard output: $x^2 + y^2$

```
>_ ~/workspace/ipp/programs
```

```
$ python3 sumofsquares.py 3 4
25
$ _
```

Integers

Program: `sumofsquares.py`

- Command-line input: x (int) and y (int)
- Standard output: $x^2 + y^2$

```
>_ ~/workspace/ipp/programs
```

```
$ python3 sumofsquares.py 3 4
25
$ python3 sumofsquares.py 6 8
```

Integers

Program: `sumofsquares.py`

- Command-line input: x (int) and y (int)
- Standard output: $x^2 + y^2$

```
>_ ~/workspace/ipp/programs
```

```
$ python3 sumofsquares.py 3 4
25
$ python3 sumofsquares.py 6 8
100
$ _
```

Integers

Integers

sumofsquares.py

```
1 import stdio
2 import sys
3
4 x = int(sys.argv[1])
5 y = int(sys.argv[2])
6 result = x * x + y * y
7 stdio.writeln(result)
```

Floats

Floats

The `float` data type represents floating-point numbers

Floats

The `float` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Floats

The `float` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Example: `3.14159` and `2.71828`

Floats

The `float` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Example: `3.14159` and `2.71828`

Scientific notation: `6.022e23` represents 6.022×10^{23} and `6.674e-11` represents 6.674×10^{-11}

Floats

The `float` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Example: `3.14159` and `2.71828`

Scientific notation: `6.022e23` represents 6.022×10^{23} and `6.674e-11` represents 6.674×10^{-11}

Operations:

Floats

The `float` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Example: `3.14159` and `2.71828`

Scientific notation: `6.022e23` represents 6.022×10^{23} and `6.674e-11` represents 6.674×10^{-11}

Operations:

- Addition (`+`)

Floats

The `float` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Example: `3.14159` and `2.71828`

Scientific notation: `6.022e23` represents 6.022×10^{23} and `6.674e-11` represents 6.674×10^{-11}

Operations:

- Addition (`+`)
- Subtraction/negation (`-`)

Floats

The `float` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Example: `3.14159` and `2.71828`

Scientific notation: `6.022e23` represents 6.022×10^{23} and `6.674e-11` represents 6.674×10^{-11}

Operations:

- Addition (`+`)
- Subtraction/negation (`-`)
- Multiplication (`*`)

Floats

The `float` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Example: `3.14159` and `2.71828`

Scientific notation: `6.022e23` represents 6.022×10^{23} and `6.674e-11` represents 6.674×10^{-11}

Operations:

- Addition (`+`)
- Subtraction/negation (`-`)
- Multiplication (`*`)
- Division (`/`)

Floats

The `float` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Example: `3.14159` and `2.71828`

Scientific notation: `6.022e23` represents 6.022×10^{23} and `6.674e-11` represents 6.674×10^{-11}

Operations:

- Addition (`+`)
- Subtraction/negation (`-`)
- Multiplication (`*`)
- Division (`/`)
- Exponentiation (`**`)

Floats

Floats

Program: `quadratic.py`

Floats

Program: `quadratic.py`

- Command-line input: a (float), b (float), and c (float)

Floats

Program: `quadratic.py`

- Command-line input: a (float), b (float), and c (float)
- Standard output: roots of the quadratic equation $ax^2 + bx + c = 0$

Floats

Program: `quadratic.py`

- Command-line input: a (float), b (float), and c (float)
- Standard output: roots of the quadratic equation $ax^2 + bx + c = 0$

```
>_ ~/workspace/ipp/programs
$ _
```

Floats

Program: `quadratic.py`

- Command-line input: a (float), b (float), and c (float)
- Standard output: roots of the quadratic equation $ax^2 + bx + c = 0$

```
>_ ~/workspace/ipp/programs
$ python3 quadratic.py 1 -5 6
```

Floats

Program: `quadratic.py`

- Command-line input: a (float), b (float), and c (float)
- Standard output: roots of the quadratic equation $ax^2 + bx + c = 0$

```
>_ ~/workspace/ipp/programs
```

```
$ python3 quadratic.py 1 -5 6
Root # 1 = 3.0
Root # 2 = 2.0
$ _
```

Floats

Program: `quadratic.py`

- Command-line input: a (float), b (float), and c (float)
- Standard output: roots of the quadratic equation $ax^2 + bx + c = 0$

```
>_ ~/workspace/ipp/programs
```

```
$ python3 quadratic.py 1 -5 6
Root # 1 = 3.0
Root # 2 = 2.0
$ python3 quadratic.py 1 -1 -1
```

Floats

Program: `quadratic.py`

- Command-line input: a (float), b (float), and c (float)
- Standard output: roots of the quadratic equation $ax^2 + bx + c = 0$

```
>_ ~/workspace/ipp/programs
```

```
$ python3 quadratic.py 1 -5 6
Root # 1 = 3.0
Root # 2 = 2.0
$ python3 quadratic.py 1 -1 -1
Root # 1 = 1.618033988749895
Root # 2 = -0.6180339887498949
$ _
```

Floats

Floats

quadratic.py

```
1  import math
2  import stdio
3  import sys
4
5  a = float(sys.argv[1])
6  b = float(sys.argv[2])
7  c = float(sys.argv[3])
8  discriminant = b * b - 4 * a * c
9  root1 = (-b + math.sqrt(discriminant)) / (2 * a)
10 root2 = (-b - math.sqrt(discriminant)) / (2 * a)
11 stdio.writeln('Root # 1 = ' + str(root1))
12 stdio.writeln('Root # 2 = ' + str(root2))
```

Booleans

Booleans

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

Booleans

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

The two `bool` literals are `True` and `False`

Booleans

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

The two `bool` literals are `True` and `False`

Operations:

Booleans

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

The two `bool` literals are `True` and `False`

Operations:

- Logical not (`not`)

Booleans

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

The two `bool` literals are `True` and `False`

Operations:

- Logical not (`not`)
- Logical or (`or`)

Booleans

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

The two `bool` literals are `True` and `False`

Operations:

- Logical not (`not`)
- Logical or (`or`)
- Logical and (`and`)

Booleans

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

The two `bool` literals are `True` and `False`

Operations:

- Logical not (`not`)
- Logical or (`or`)
- Logical and (`and`)

Truth tables for the logical operations

x	not x
False	True
True	False

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

Booleans

Booleans

Two objects of the same type can be compared using comparison operators — the result is a boolean value

Booleans

Two objects of the same type can be compared using comparison operators — the result is a boolean value

Comparison operators:

Booleans

Two objects of the same type can be compared using comparison operators — the result is a boolean value

Comparison operators:

- Equal (==)

Booleans

Two objects of the same type can be compared using comparison operators — the result is a boolean value

Comparison operators:

- Equal (`==`)
- Not equal (`!=`)

Booleans

Two objects of the same type can be compared using comparison operators — the result is a boolean value

Comparison operators:

- Equal (`==`)
- Not equal (`!=`)
- Less than (`<`)

Booleans

Two objects of the same type can be compared using comparison operators — the result is a boolean value

Comparison operators:

- Equal (`==`)
- Not equal (`!=`)
- Less than (`<`)
- Less than or equal (`<=`)

Booleans

Two objects of the same type can be compared using comparison operators — the result is a boolean value

Comparison operators:

- Equal (`==`)
- Not equal (`!=`)
- Less than (`<`)
- Less than or equal (`<=`)
- Greater than (`>`)

Booleans

Two objects of the same type can be compared using comparison operators — the result is a boolean value

Comparison operators:

- Equal (`==`)
- Not equal (`!=`)
- Less than (`<`)
- Less than or equal (`<=`)
- Greater than (`>`)
- Greater than or equal (`>=`)

Booleans

Booleans

Program: `leapyear.py`

Booleans

Program: `leapyear.py`

- Command-line input: y (int)

Booleans

Program: `leapyear.py`

- Command-line input: y (int)
- Standard output: whether y is a leap year or not

Booleans

Program: `leapyear.py`

- Command-line input: y (int)
- Standard output: whether y is a leap year or not

```
>_ ~/workspace/ipp/programs
$ _
```

Booleans

Program: `leapyear.py`

- Command-line input: y (int)
- Standard output: whether y is a leap year or not

```
>_ ~/workspace/ipp/programs
$ python3 leapyear.py 2020
```

Booleans

Program: `leapyear.py`

- Command-line input: y (int)
- Standard output: whether y is a leap year or not

```
>_ ~/workspace/ipp/programs

$ python3 leapyear.py 2020
True
$ _
```

Booleans

Program: `leapyear.py`

- Command-line input: y (int)
- Standard output: whether y is a leap year or not

```
>_ ~/workspace/ipp/programs
```

```
$ python3 leapyear.py 2020
True
$ python3 leapyear.py 1900
```

Booleans

Program: `leapyear.py`

- Command-line input: y (int)
- Standard output: whether y is a leap year or not

```
>_ ~/workspace/ipp/programs
```

```
$ python3 leapyear.py 2020
True
$ python3 leapyear.py 1900
False
$ _
```

Booleans

Program: `leapyear.py`

- Command-line input: y (int)
- Standard output: whether y is a leap year or not

```
>_ ~/workspace/ipp/programs
```

```
$ python3 leapyear.py 2020
True
$ python3 leapyear.py 1900
False
$ python3 leapyear.py 2000
```

Booleans

Program: `leapyear.py`

- Command-line input: y (int)
- Standard output: whether y is a leap year or not

```
>_ ~/workspace/ipp/programs
```

```
$ python3 leapyear.py 2020
True
$ python3 leapyear.py 1900
False
$ python3 leapyear.py 2000
True
$ _
```

Booleans

Booleans

leapyear.py

```
1  import stdio
2  import sys
3
4  y = int(sys.argv[1])
5  result = y % 4 == 0 and y % 100 != 0 or y % 400 == 0
6  stdio.writeln(result)
```

Operator Precedence

Operator Precedence

Operator precedence (highest to lowest)

`**`	exponentiation
`+, -`	unary
`*, /, //, %`	multiplicative
`+, -`	additive
`<, <=, >, >=`	comparison
`==, !=`	equality
`=, **=, *=, /=, //=, %=, +=, -=`	assignment
`is, is not`	identity
`in, not in`	membership
`not, or, and`	logical

Operator Precedence

Operator precedence (highest to lowest)

`**`	exponentiation
`+, -`	unary
`*, /, //, %`	multiplicative
`+, -`	additive
`<, <=, >, >=`	comparison
`==, !=`	equality
`=, **=, *=, /=, //=, %=, +=, -=`	assignment
`is, is not`	identity
`in, not in`	membership
`not, or, and`	logical

Parentheses can be used to override precedence rules

Python Console

Python Console

The Python Console[1] available in PyCharm can be used as an interactive calculator

[1] To launch from terminal, run the command `python3`; and to return to the terminal, run the built-in function `exit()`

Python Console

The Python Console[1] available in PyCharm can be used as an interactive calculator

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> _
```

[1] To launch from terminal, run the command `python3`; and to return to the terminal, run the built-in function `exit()`

Python Console

The Python Console[1] available in PyCharm can be used as an interactive calculator

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> 3 ** 2 + 4 ** 2
```

[1] To launch from terminal, run the command `python3`; and to return to the terminal, run the built-in function `exit()`

Python Console

The Python Console[1] available in PyCharm can be used as an interactive calculator

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> 3 ** 2 + 4 ** 2
25
>>> _
```

[1] To launch from terminal, run the command `python3`; and to return to the terminal, run the built-in function `exit()`

Python Console

The Python Console[1] available in PyCharm can be used as an interactive calculator

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> 3 ** 2 + 4 ** 2
25
>>> import math
```

[1] To launch from terminal, run the command `python3`; and to return to the terminal, run the built-in function `exit()`

Python Console

The Python Console[1] available in PyCharm can be used as an interactive calculator

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> 3 ** 2 + 4 ** 2
25
>>> import math
>>> _
```

[1] To launch from terminal, run the command `python3`; and to return to the terminal, run the built-in function `exit()`

Python Console

The Python Console[1] available in PyCharm can be used as an interactive calculator

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> 3 ** 2 + 4 ** 2
25
>>> import math
>>> x = 2
```

[1] To launch from terminal, run the command `python3`; and to return to the terminal, run the built-in function `exit()`

Python Console

The Python Console[1] available in PyCharm can be used as an interactive calculator

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> 3 ** 2 + 4 ** 2
25
>>> import math
>>> x = 2
>>> _
```

[1] To launch from terminal, run the command `python3`; and to return to the terminal, run the built-in function `exit()`

Python Console

The Python Console[1] available in PyCharm can be used as an interactive calculator

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> 3 ** 2 + 4 ** 2
25
>>> import math
>>> x = 2
>>> math.sqrt(x)
```

[1] To launch from terminal, run the command `python3`; and to return to the terminal, run the built-in function `exit()`

Python Console

The Python Console[1] available in PyCharm can be used as an interactive calculator

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> 3 ** 2 + 4 ** 2
25
>>> import math
>>> x = 2
>>> math.sqrt(x)
1.4142135623730951
>>> _
```

[1] To launch from terminal, run the command `python3`; and to return to the terminal, run the built-in function `exit()`

Python Console

Python Console

Run `dir(<library>)` to get a list of attributes for a library

Python Console

Run `dir(<library>)` to get a list of attributes for a library

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> _
```

Python Console

Run `dir(<library>)` to get a list of attributes for a library

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> dir(math)
```

Python Console

Run `dir(<library>)` to get a list of attributes for a library

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>> _
```

Python Console

Python Console

Run `help(<library>)` to access documentation for a library

Python Console

Run `help(<library>)` to access documentation for a library

Example

```
>_ ~/workspace/ipp/programs
>>> _
```

Python Console

Run `help(<library>)` to access documentation for a library

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> help(math)
```

Python Console

Run `help(<library>)` to access documentation for a library

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> 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
>>> _
```

Python Console

Python Console

Run `help(<library>.<name>)` to access documentation for a particular function from a library

Python Console

Run `help(<library>.<name>)` to access documentation for a particular function from a library

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> _
```

Python Console

Run `help(<library>.<name>)` to access documentation for a particular function from a library

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> help(math.sqrt)
```

Python Console

Run `help(<library>.<name>)` to access documentation for a particular function from a library

Example

```
>_ ~/workspace/ipp/programs
```

```
>>> help(math.sqrt)
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
>>> _
```