Introduction to Programming in Python

Imperative Programming: Basic Data Types

Outline

Data Types

Data Types

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

Data Types

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

Basic data types

- `str` for sequences of characters with string operations
- `int` for integers with arithmetic operations
- `float` for floating-point numbers (aka floats) with arithmetic operations
- `bool` for boolean (true/false) values with logical operations
- `complex` for complex numbers specified as `<real part> + <imaginary part>j` (eg, `2 + 3j`) with complex-number operations

Expressions

Expressions

A literal represents a basic data-type value

Expressions

A literal represents a basic data-type value

Example

- `"Hello, World"` is a string literal
- `42` is an integer literal
- `3.14159` is a floating-point literal
- `True` and `False` are boolean literals

Expressions

Expressions

An identifier represents a name

Expressions

An identifier represents a name

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

Expressions

An identifier represents a name

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

Example: `abc`, `_abc`, `abc123`, and `Abc` are valid identifiers whereas `abc*`, `1abc`, and `abc+` are not

Expressions

An identifier represents a name

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

Example: `abc`, `_abc`, `abc123`, and `Abc` are valid identifiers whereas `abc*`, `1abc`, and `abc+` are not

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

Expressions

Expressions

A variable associates a name with a data-type value

Expressions

A variable associates a name with a data-type value

Example: age

Expressions

A variable associates a name with a data-type value

Example: age

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

Expressions

A variable associates a name with a data-type value

Example: age

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

Example: `SPEED_OF_LIGHT`

Expressions

A variable associates a name with a data-type value

Example: `age`

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

Example: `SPEED_OF_LIGHT`

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

Expressions

A variable associates a name with a data-type value

Example: `age`

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

Example: `SPEED_OF_LIGHT`

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

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

Expressions

Expressions

An operator represents a data-type operation

Expressions

An operator represents a data-type operation

Example

- `+`, `-`, `*`, `/`, `%`, 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 operators, but also functions

Expressions

Many programming tasks involve not only 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 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 `<name>(<arg1>, <arg2>, ...)` or `<target>.<name>(<arg1>, <arg2>, ...)`

Expressions

Many programming tasks involve not only 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 `<name>(<arg1>, <arg2>, ...)` or `<target>.<name>(<arg1>, <arg2>, ...)`

Example: `stdio.writeln("Hello, World")` and `math.sqrt(2)`

Expressions

Many programming tasks involve not only 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 `<name>(<arg1>, <arg2>, ...)` or `<target>.<name>(<arg1>, <arg2>, ...)`

Example: `stdio.writeln("Hello, World")` and `math.sqrt(2)`

A function that does not return a value is called a void function (eg, `stdio.writeln()`)

Expressions

Many programming tasks involve not only 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 `<name>(<arg1>, <arg2>, ...)` or `<target>.<name>(<arg1>, <arg2>, ...)`

Example: `stdio.writeln("Hello, World")` and `math.sqrt(2)`

A function that does not return a value is called a void function (eg, `stdio.writeln()`)

A function that returns a value is called a non-void function (eg, `math.sqrt()`)

Expressions

Expressions

Example

Built-in Functions

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

`sys`

`exit(x = "")`	exits the Python interpreter with the message `x`

Expressions

Example

`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

Expressions

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

Every expression has a type and a value

Expressions

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

Every expression has a type and 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 syntactic unit that expresses some action to be carried out

Statements

A syntactic unit that expresses some action to be carried out

Example

```
import stdio
import sys

message = sys.argv[1]
stdio.writeln(message)
```

Statements

Statements

Import statement

```
import <library>
```

Statements

Import statement

```
import <library>
```

Example

```
import math
import stdio
import sys
```

Statements

Statements

Function call statement

```
<name>(<arg1>, <arg2>, ...)
<target>.<name>(<arg1>, <arg2>, ...)
```

Statements

Function call statement

```
<name>(<arg1>, <arg2>, ...)
<target>.<name>(<arg1>, <arg2>, ...)
```

Example

```
stdio.writeln("To be, or not to be, that is the question.")
sys.exit("Done!")
```

Statements

Assignment statement

```
<name> = <expression>
<name1>, <name2>, <name3>, ... = <expression1>, <expression2>, <expression3>, ...
```

Statements

Assignment statement

```
<name> = <expression>
<name1>, <name2>, <name3>, ... = <expression1>, <expression2>, <expression3>, ...
```

Example

```python
a = "Python"
b, c, d = 42, 3.14159, True
e = None
```

Statements

Statements

Example (swapping idiom)

```
import stdio

a = 42
b = 1729

t = a
a = b
b = t

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

line #	a	b	t

Statements

Example (swapping idiom)

```
1  import stdio
2
3  a = 42
4  b = 1729
5
6  t = a
7  a = b
8  b = t
9
10 stdio.writeln(a)
11 stdio.writeln(b)
```

line #	a	b	t
1			

Statements

Example (swapping idiom)

```
import stdio

a = 42
b = 1729

t = a
a = b
b = t

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

line #	a	b	t
3	42		

×

Statements

Example (swapping idiom)

```
1  import stdio
2
3  a = 42
4  b = 1729
5
6  t = a
7  a = b
8  b = t
9
10 stdio.writeln(a)
11 stdio.writeln(b)
```

line #	a	b	t
4	42	1729	

Example (swapping idiom)

```
1  import stdio
2
3  a = 42
4  b = 1729
5
6  t = a
7  a = b
8  b = t
9
10 stdio.writeln(a)
11 stdio.writeln(b)
```

line #	a	b	t
6	42	1729	42

Example (swapping idiom)

```
1  import stdio
2
3  a = 42
4  b = 1729
5
6  t = a
7  a = b
8  b = t
9
10 stdio.writeln(a)
11 stdio.writeln(b)
```

line #	a	b	t
7	1729	1729	42

Example (swapping idiom)

```
import stdio

a = 42
b = 1729

t = a
a = b
b = t

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

line #	a	b	t
8	1729	42	42

×

1
2

Statements

Example (swapping idiom)

```
1  import stdio
2
3  a = 42
4  b = 1729
5
6  t = a
7  a = b
8  b = t
9
10 stdio.writeln(a)
11 stdio.writeln(b)
```

line #	a	b	t
10	1729	42	42

```
1729
```

Statements

Example (swapping idiom)

```
1  import stdio
2
3  a = 42
4  b = 1729
5
6  t = a
7  a = b
8  b = t
9
10 stdio.writeln(a)
11 stdio.writeln(b)
```

line #	a	b	t
11	1729	42	42

```
1729
42
```

Statements

Example (swapping idiom)

```
import stdio

a = 42
b = 1729

t = a
a = b
b = t

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

line #	a	b	t

```
1729
42
```

Statements

Statements

Example (variable update)

```
import stdio

x = 2
x = x ** 5
x = x * 2
x = x / 4
x = x // 3
x = x % 3
x = x + 1
x = x - 1

stdio.writeln(x)
```

line #	x

Example (variable update)

```
1  import stdio
2
3  x = 2
4  x = x ** 5
5  x = x * 2
6  x = x / 4
7  x = x // 3
8  x = x % 3
9  x = x + 1
10 x = x - 1
11
12 stdio.writeln(x)
```

line #	x
1	

Example (variable update)

```
1  import stdio
2
3  x = 2
4  x = x ** 5
5  x = x * 2
6  x = x / 4
7  x = x // 3
8  x = x % 3
9  x = x + 1
10 x = x - 1
11
12 stdio.writeln(x)
```

line #	x
3	2

×

Statements

Example (variable update)

```
1  import stdio
2
3  x = 2
4  x = x ** 5
5  x = x * 2
6  x = x / 4
7  x = x // 3
8  x = x % 3
9  x = x + 1
10 x = x - 1
11
12 stdio.writeln(x)
```

line #	x
4	32

Statements

Example (variable update)

```
1  import stdio
2
3  x = 2
4  x = x ** 5
5  x = x * 2
6  x = x / 4
7  x = x // 3
8  x = x % 3
9  x = x + 1
10 x = x - 1
11
12 stdio.writeln(x)
```

line #	x
5	64

Statements

Example (variable update)

```
1  import stdio
2
3  x = 2
4  x = x ** 5
5  x = x * 2
6  x = x / 4
7  x = x // 3
8  x = x % 3
9  x = x + 1
10 x = x - 1
11
12 stdio.writeln(x)
```

line #	x
6	16.0

Statements

Example (variable update)

```
import stdio

x = 2
x = x ** 5
x = x * 2
x = x / 4
x = x // 3
x = x % 3
x = x + 1
x = x - 1

stdio.writeln(x)
```

line #	x
7	5.0

Example (variable update)

```
1  import stdio
2
3  x = 2
4  x = x ** 5
5  x = x * 2
6  x = x / 4
7  x = x // 3
8  x = x % 3
9  x = x + 1
10 x = x - 1
11
12 stdio.writeln(x)
```

line #	x
8	2.0

Statements

Example (variable update)

```
1  import stdio
2
3  x = 2
4  x = x ** 5
5  x = x * 2
6  x = x / 4
7  x = x // 3
8  x = x % 3
9  x = x + 1
10 x = x - 1
11
12 stdio.writeln(x)
```

line #	x
9	3.0

Statements

Example (variable update)

```
1  import stdio
2
3  x = 2
4  x = x ** 5
5  x = x * 2
6  x = x / 4
7  x = x // 3
8  x = x % 3
9  x = x + 1
10 x = x - 1
11
12 stdio.writeln(x)
```

line #	x
10	2.0

Statements

Example (variable update)

```
1 import stdio
2
3 x = 2
4 x = x ** 5
5 x = x * 2
6 x = x / 4
7 x = x // 3
8 x = x % 3
9 x = x + 1
10 x = x - 1
11
12 stdio.writeln(x)
```

line #	x
12	2.0

```
2.0
```

Statements

Example (variable update)

```
import stdio

x = 2
x = x ** 5
x = x * 2
x = x / 4
x = x // 3
x = x % 3
x = x + 1
x = x - 1

stdio.writeln(x)
```

line #	x

```
2.0
```

Statements

Statements

The assignment statement

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

is equivalent to

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

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

Statements

Statements

Example

```
x = x ** 5
x = x * 2
x = x / 4
x = x // 3
x = x % 3
x = x + 1
x = x - 1
```

are equivalent to

```
x **= 5
x *= 2
x /= 4
x //= 3
x %= 3
x += 1
x -= 1
```

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 double quotes

Strings

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

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

Example: `"Hello, World"`

Strings

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

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

Example: `"Hello, World"`

Tab, newline, backslash, and double 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 double quotes

Example: `"Hello, World"`

Tab, newline, backslash, and double quote characters are specified using escape sequences `"\t"`, `"\n"`, `"\\"`, and `"\""`

Example: `"Hello, world\n"`

Strings

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

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

Example: `"Hello, World"`

Tab, newline, backslash, and double quote characters are specified using escape sequences `"\t"`, `"\n"`, `"\\"`, and `"\""`

Example: `"Hello, world\n"`

Operations

- Concatenation (+) — eg, `"123" + "456"` evaluates to `"123456"`
- Replication (*) — eg, `3 * "ab"` and `"ab" * 3` evaluate to `"ababab"`

Strings

`dateformats.py`

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

`dateformats.py`

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

```
× ~/workspace/ipp

$ _
```

Strings

`dateformats.py`

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

~/workspace/ipp

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

`dateformats.py`

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

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

Strings

dateformats.py

```
import stdio
import sys

d = sys.argv[1]
m = sys.argv[2]
y = sys.argv[3]
dmy = d + "/" + m + "/" + y
mdy = m + "/" + d + "/" + y
ymd = y + "/" + m + "/" + d
stdio.writeln(dmy)
stdio.writeln(mdy)
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`

Integers

The `int` data type represents integers

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

Example: `42`

Operations

- Addition (+) — eg, `5 + 2` evaluates to `7`
- Subtraction/negation (-) — eg, `5 - 2` evaluates to `3` and `-(-3)` evaluates to `3`
- Multiplication (*) — eg, `5 * 2` evaluates to `10`
- Division (/) — eg, `5 / 2` evaluates to `2.5`
- Floored division (//) — eg, `5 // 2` evaluates to `2`
- Remainder (%) — eg, `5 % 2` evaluates to `1`
- Exponentiation (**) — eg, `5 ** 2` evaluates to `25`

Integers

Integers

`sumofsquares.py`

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

Integers

`sumofsquares.py`

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

```
× ~/workspace/ipp
$ _
```

`sumofsquares.py`

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

```
× ~/workspace/ipp
$ python3 sumofsquares.py 3 4
```

`sumofsquares.py`

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

```
× ~/workspace/ipp

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

Integers

`sumofsquares.py`

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

```
× ~/workspace/ipp
$ python3 sumofsquares.py 3 4
25
$ python3 sumofsquares.py 6 8
```

Integers

`sumofsquares.py`

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

```
× ~/workspace/ipp
$ python3 sumofsquares.py 3 4
25
$ python3 sumofsquares.py 6 8
100
$ _
```

Integers

Integers

sumofsquares.py

```
import stdio
import sys

x = int(sys.argv[1])
y = int(sys.argv[2])
result = x * x + y * y
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`

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`

Scientific notation can be used to represent very large and very small 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

Example: `3.14159`

Scientific notation can be used to represent very large and very small numbers

Example: `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`

Scientific notation can be used to represent very large and very small numbers

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

Operations

- Addition (+) — eg, `16.0 + 0.5` evaluates to `16.5`
- Subtraction/negation (-) — eg, `16.0 - 0.5` evaluates to `15.5` and `-(-3.0)` evaluates to `3.0`
- Multiplication (*) — eg, `16.0 * 0.5` evaluates to `8.0`
- Division (/) — eg, `16.0 / 0.5` evaluates to `32.0`
- Exponentiation (**) — eg, `16.0 ** 0.5` evaluates to `4.0`

Floats

Floats

`quadratic.py`

- Command-line input: a (float), b (float), and c (float)
- Standard output: the two roots of the quadratic equation $ax^2 + bx + c = 0$, computed as $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Floats

`quadratic.py`

- Command-line input: a (float), b (float), and c (float)
- Standard output: the two roots of the quadratic equation $ax^2 + bx + c = 0$, computed as $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

```
~/workspace/ipp
$ _
```

Floats

`quadratic.py`

- Command-line input: a (float), b (float), and c (float)
- Standard output: the two roots of the quadratic equation $ax^2 + bx + c = 0$, computed as $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

```
× ~/workspace/ipp
$ python3 quadratic.py 1 -5 6
```

Floats

quadratic.py

- Command-line input: a (float), b (float), and c (float)
- Standard output: the two roots of the quadratic equation $ax^2 + bx + c = 0$, computed as $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

~/workspace/ipp

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

Floats

quadratic.py

- Command-line input: a (float), b (float), and c (float)
- Standard output: the two roots of the quadratic equation $ax^2 + bx + c = 0$, computed as $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

```
× ~/workspace/ipp
```

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

Floats

quadratic.py

- Command-line input: a (float), b (float), and c (float)
- Standard output: the two roots of the quadratic equation $ax^2 + bx + c = 0$, computed as $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

~/workspace/ipp

```
$ 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

```
import math
import stdio
import sys

a = float(sys.argv[1])
b = float(sys.argv[2])
c = float(sys.argv[3])
discriminant = b * b - 4 * a * c
root1 = (-b + math.sqrt(discriminant)) / (2 * a)
root2 = (-b - math.sqrt(discriminant)) / (2 * a)
stdio.writeln("Root # 1 = " + str(root1))
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

- 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 of which is a boolean value

Booleans

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

Comparison operators

- Equal (==) — eg, `5 == 2` evaluates to `False`
- Not equal (`!=`) — eg, `5 != 2` evaluates to `True`
- Less than (<) — eg, `5 < 2` evaluates to `False`
- Less than or equal (<=) — eg, `5 <= 2` evaluates to `False`
- Greater than (>) — eg, `5 > 2` evaluates to `True`
- Greater than or equal (>=) — eg, `5 >= 2` evaluates to `True`

Booleans

Booleans

`leapyear.py`

- Command-line input: y (int)
- Standard output: `True` if y is a leap year and `False` otherwise

Booleans

`leapyear.py`

- Command-line input: y (int)
- Standard output: `True` if y is a leap year and `False` otherwise

```
× ~/workspace/ipp

$ _
```

Booleans

`leapyear.py`

- Command-line input: y (int)
- Standard output: `True` if y is a leap year and `False` otherwise

```
~/workspace/ipp

$ python3 leapyear.py 2020
```

Booleans

`leapyear.py`

- Command-line input: y (int)
- Standard output: `True` if y is a leap year and `False` otherwise

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

Booleans

`leapyear.py`

- Command-line input: y (int)
- Standard output: `True` if y is a leap year and `False` otherwise

```
~/workspace/ipp
$ python3 leapyear.py 2020
True
$ python3 leapyear.py 1900
```

`leapyear.py`

- Command-line input: y (int)
- Standard output: `True` if y is a leap year and `False` otherwise

```
~/workspace/ipp

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

Booleans

`leapyear.py`

- Command-line input: y (int)
- Standard output: `True` if y is a leap year and `False` otherwise

```
× ~/workspace/ipp
$ python3 leapyear.py 2020
True
$ python3 leapyear.py 1900
False
$ python3 leapyear.py 2000
```

Booleans

`leapyear.py`

- Command-line input: y (int)
- Standard output: `True` if y is a leap year and `False` otherwise

```
× ~/workspace/ipp
$ python3 leapyear.py 2020
True
$ python3 leapyear.py 1900
False
$ python3 leapyear.py 2000
True
$ _
```

`leapyear.py`

- Command-line input: y (int)
- Standard output: `True` if y is a leap year and `False` otherwise

```
~/workspace/ipp
$ python3 leapyear.py 2020
True
$ python3 leapyear.py 1900
False
$ python3 leapyear.py 2000
True
$ _
```

A leap year is one that is divisible by 4 *and* is not divisible by 100 *or* is divisible by 400

Booleans

Booleans

leapyear.py

```
import stdio
import sys

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

Operator Precedence

Operator Precedence

From highest to lowest

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

Operator Precedence

From highest to lowest

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

Example: `3 + 2 ** 3` evaluates to `11`

Operator Precedence

From highest to lowest

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

Example: `3 + 2 ** 3` evaluates to `11`

Parentheses can be used to override precedence rules

Operator Precedence

From highest to lowest

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

Example: `3 + 2 ** 3` evaluates to `11`

Parentheses can be used to override precedence rules

Example: `(3 + 2) ** 3` evaluates to `125`