Basic Python Datatypes

Basic Python Datatypes

How to determine and manipulate the `type` in Python

List of Basic Datatypes

List of Basic Datatypes

We have already seen three of the basic data types in Python:

List of Basic Datatypes

We have already seen three of the basic data types in Python:

- Integers (`int type`)

List of Basic Datatypes

We have already seen three of the basic data types in Python:

- Integers (`int type`)
  - $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$

List of Basic Datatypes

We have already seen three of the basic data types in Python:

- Integers (`int type`)
  - $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$
- Floating-point (decimal/fractional) numbers (`float type`)

List of Basic Datatypes

We have already seen three of the basic data types in Python:

- Integers (`int type`)
  - $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$
- Floating-point (decimal/fractional) numbers (`float type`)
  - $1.5, -1.5, 1.0, ...$

List of Basic Datatypes

We have already seen three of the basic data types in Python:

- Integers (`int type`)
  - $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$
- Floating-point (decimal/fractional) numbers (`float type`)
  - $1.5, -1.5, 1.0, ...$
- Strings (words and sentences) (`str type`)

List of Basic Datatypes

We have already seen three of the basic data types in Python:

- Integers (`int type`)
  - $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$
- Floating-point (decimal/fractional) numbers (`float type`)
  - $1.5, -1.5, 1.0, ...$
- Strings (words and sentences) (`str type`)
  - `"Tyler"`, `"hello world"`, `"%#()*"`, `"1.0"`

List of Basic Datatypes

We have already seen three of the basic data types in Python:

- Integers (`int type`)
  - $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$
- Floating-point (decimal/fractional) numbers (`float type`)
  - $1.5, -1.5, 1.0, ...$
- Strings (words and sentences) (`str type`)
  - `"Tyler"`, `"hello world"`, `"%#()*"`, `"1.0"`

One more type in the future:

List of Basic Datatypes

We have already seen three of the basic data types in Python:

- Integers (`int type`)
  - $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$
- Floating-point (decimal/fractional) numbers (`float type`)
  - $1.5, -1.5, 1.0, ...$
- Strings (words and sentences) (`str type`)
  - `"Tyler"`, `"hello world"`, `"%#()*"`, `"1.0"`

One more type in the future:

- Boolean (`bool type`)

List of Basic Datatypes

We have already seen three of the basic data types in Python:

- Integers (`int type`)
  - $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$
- Floating-point (decimal/fractional) numbers (`float type`)
  - $1.5, -1.5, 1.0, ...$
- Strings (words and sentences) (`str type`)
  - `"Tyler"`, `"hello world"`, `"%#()*"`, `"1.0"`

One more type in the future:

- Boolean (`bool type`)
  - `True`, `False`

Numeric operators

Numeric operators

The *numeric* types are the `float` and `int` types.

Numeric operators

The *numeric* types are the `float` and `int` types.

Use the following operators to calculate with numeric types:

Numeric operators

The *numeric* types are the `float` and `int` types.

Use the following operators to calculate with numeric types:

- `**` (exponent)
- `*` (multiplication)
- `/` (division)
- `//` (integer division)
- `%` (remainder)
- `+` (addition)
- `-` (subtraction)

Numeric operators

The *numeric* types are the `float` and `int` types.

Use the following operators to calculate with numeric types:

- `**` (exponent)
- `*` (multiplication)
- `/` (division)
- `//` (integer division)
- `%` (remainder)
- `+` (addition)
- `-` (subtraction)

Remember: Python follows the PEMDAS order of operations

Numeric operators

The *numeric* types are the `float` and `int` types.

Use the following operators to calculate with numeric types:

- `**` (exponent)
- `*` (multiplication)
- `/` (division)
- `//` (integer division)
- `%` (remainder)
- `+` (addition)
- `-` (subtraction)

Remember: Python follows the PEMDAS order of operations

Use parentheses `( ... )` to control what happens first

How Operators Affect Type

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- `>>> 2 * (3 + 1) - 7.0`

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- ```
  >>> 2 * (3 + 1) - 7.0
  1.0
  ```
  ```

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- ```
  >>> 2 * (3 + 1) - 7.0
  1.0
  ```

If all types are `int`, then the result will also be an `int`:
  ```

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- ```
  >>> 2 * (3 + 1) - 7.0
  1.0
  ```

If all types are `int`, then the result will also be an `int`:
  ```

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- `>>> 2 * (3 + 1) - 7.0`
  `1.0`

If all types are `int`, then the result will also be an `int`:

- `>>> 2 * (3 + 1) - 7`

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- ```
  >>> 2 * (3 + 1) - 7.0
  1.0
  ```

If all types are `int`, then the result will also be an `int`:

- ```
  >>> 2 * (3 + 1) - 7
  1
  ```

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- ```
  >>> 2 * (3 + 1) - 7.0
  1.0
  ```

If all types are `int`, then the result will also be an `int`:

- ```
  >>> 2 * (3 + 1) - 7
  1
  ```

Unless there is a *division* or *negative exponent* involved:

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- ```
  >>> 2 * (3 + 1) - 7.0
  1.0
  ```

If all types are `int`, then the result will also be an `int`:

- ```
  >>> 2 * (3 + 1) - 7
  1
  ```

Unless there is a *division* or *negative exponent* involved:

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- ```
  >>> 2 * (3 + 1) - 7.0
  1.0
  ```

If all types are `int`, then the result will also be an `int`:

- ```
  >>> 2 * (3 + 1) - 7
  1
  ```

Unless there is a *division* or *negative exponent* involved:

- ```
  >>> 2 * (3 + 1) / 8
  ```
  ```

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- ```
  >>> 2 * (3 + 1) - 7.0
  1.0
  ```

If all types are `int`, then the result will also be an `int`:

- ```
  >>> 2 * (3 + 1) - 7
  1
  ```

Unless there is a *division* or *negative exponent* involved:

- ```
  >>> 2 * (3 + 1) / 8
  1.0
  ```
  ```

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- ```
  >>> 2 * (3 + 1) - 7.0
  1.0
  ```

If all types are `int`, then the result will also be an `int`:

- ```
  >>> 2 * (3 + 1) - 7
  1
  ```

Unless there is a *division* or *negative exponent* involved:

- ```
  >>> 2 * (3 + 1) / 8
  1.0
  ```

To round down to the nearest `int`, use the *integer divison* operator:
  ```

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- ```
  >>> 2 * (3 + 1) - 7.0
  1.0
  ```

If all types are `int`, then the result will also be an `int`:

- ```
  >>> 2 * (3 + 1) - 7
  1
  ```

Unless there is a *division* or *negative exponent* involved:

- ```
  >>> 2 * (3 + 1) / 8
  1.0
  ```

To round down to the nearest `int`, use the *integer divison* operator:
  ```

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- ```
  >>> 2 * (3 + 1) - 7.0
  1.0
  ```

If all types are `int`, then the result will also be an `int`:

- ```
  >>> 2 * (3 + 1) - 7
  1
  ```

Unless there is a *division* or *negative exponent* involved:

- ```
  >>> 2 * (3 + 1) / 8
  1.0
  ```

To round down to the nearest `int`, use the *integer divison* operator:

- ```
  >>> 3 // 2
  ```

How Operators Affect Type

A numeric calculation involving a `float` will *promote* the result:

- ```
  >>> 2 * (3 + 1) - 7.0
  1.0
  ```

If all types are `int`, then the result will also be an `int`:

- ```
  >>> 2 * (3 + 1) - 7
  1
  ```

Unless there is a *division* or *negative exponent* involved:

- ```
  >>> 2 * (3 + 1) / 8
  1.0
  ```

To round down to the nearest `int`, use the *integer divison* operator:

- ```
  >>> 3 // 2
  1
  ```

String operators

String operators

The `str` type supports only one operator:

String operators

The `str` type supports only one operator:

- `+` concatenates two strings

String operators

The `str` type supports only one operator:

- `+` concatenates two strings

```
>>> "Hello" + "World"
"HelloWorld"
```

String operators

The `str` type supports only one operator:

- `+` concatenates two strings

```
>>> "Hello" + "World"
"HelloWorld"
```

If you place two strings next to each other separated only by whitespace, then Python uses the `+` operator implicitly:

String operators

The `str` type supports only one operator:

- `+` concatenates two strings

```
>>> "Hello" + "World"
"HelloWorld"
```

If you place two strings next to each other separated only by whitespace, then Python uses the `+` operator implicitly:

```
>>> "Hello" "World"
"HelloWorld"
```

String operators

String operators

Python does **not** support any other numeric operators for the `str` type:

String operators

Python does **not** support any other numeric operators for the `str` type:

```
>>> "Hello" * "World"
TypeError:  can't multiply sequence by type "str"
```

String operators

Python does **not** support any other numeric operators for the `str` type:

```
>>> "Hello" * "World"
TypeError:  can't multiply sequence by type "str"
```

Also, you can **not** concatenate a `str` with a numeric type by using the + operator:

String operators

Python does **not** support any other numeric operators for the `str` type:

```
>>> "Hello" * "World"
TypeError:  can't multiply sequence by type "str"
```

Also, you can **not** concatenate a `str` with a numeric type by using the + operator:

```
>>> "The answer is:  " + 5
TypeError:  can only concatenate str (not "int") to
str
```

Concept Check!

What is the result and datatype for each of the following Python commands:

- `>>> (1 + 1) * 2`
- `>>> (1 + 1) * 2.0`
- `>>> (2 + 1) / 2`
- `>>> (2 + 1) // 2`
- `>>> (1 + 1) ** 2`
- `>>> (1 + 1) ** -2`
- `>>> "hello" + " " + "world"`
- `>>> "hello" " " "world"`
- `>>> "I have " + 5 + " dollars"`
- `>>> "I have " + "5" + " dollars"`

Organizing Your Calculations in a Script

Organizing Your Calculations in a Script

Sometimes, you will need to do long complicated calculations:

Organizing Your Calculations in a Script

Sometimes, you will need to do long complicated calculations:

```
x = (5 * 6) ** 2 - 10 / (2 ** (3 + 5)) + 67.7
```

Organizing Your Calculations in a Script

Sometimes, you will need to do long complicated calculations:

```
x = (5 * 6) ** 2 - 10 / (2 ** (3 + 5)) + 67.7
```

Split these onto multiple lines using a continuation character:

Organizing Your Calculations in a Script

Sometimes, you will need to do long complicated calculations:

```
x = (5 * 6) ** 2 - 10 / (2 ** (3 + 5)) + 67.7
```

Split these onto multiple lines using a continuation character:

```
x = (5 * 6) ** 2 - \
    10 / (2 ** (3 + 5)) + 67.7
```

Organizing Your Calculations in a Script

Sometimes, you will need to do long complicated calculations:

```
x = (5 * 6) ** 2 - 10 / (2 ** (3 + 5)) + 67.7
```

Split these onto multiple lines using a continuation character:

```
x = (5 * 6) ** 2 - \
    10 / (2 ** (3 + 5)) + 67.7
```

Python will not run a line with open parentheses, so you can achieve the same thing using parentheses:

Organizing Your Calculations in a Script

Sometimes, you will need to do long complicated calculations:

```
x = (5 * 6) ** 2 - 10 / (2 ** (3 + 5)) + 67.7
```

Split these onto multiple lines using a continuation character:

```
x = (5 * 6) ** 2 - \
    10 / (2 ** (3 + 5)) + 67.7
```

Python will not run a line with open parentheses, so you can achieve the same thing using parentheses:

```
x = ((5 * 6) ** 2 -
       10 / (2 ** (3 + 5)) + 67.7)
```

Organizing Your Calculations in a Script

Organizing Your Calculations in a Script

Also, use comments to document complicated blocks of code:

Organizing Your Calculations in a Script

Also, use comments to document complicated blocks of code:

```
PI = 3.14
# Get inputs from the user
rad = float(input())
height = float(input())
# Calculate the volume of a cylinder
base = rad ** 2 * PI
vol = base * height
# Print the output
print("The volume is ", vol)
```

Keeping track of the type

Keeping track of the type

Oops, I forgot what the type is!

Keeping track of the type

Oops, I forgot what the type is!

```
x = 3.0
x = x + 1
...
```

Keeping track of the type

Oops, I forgot what the type is!

```
x = 3.0
x = x + 1
...
x = ?
```

Keeping track of the type

Oops, I forgot what the type is!

```
x = 3.0
x = x + 1
...
print(type(x))
```

Keeping track of the type

Oops, I forgot what the type is!

```
x = 3.0
x = x + 1
...
print(type(x))
```

```
<class "float">
```

Type casting

Type casting

Not happy with your types?

Type casting

Not happy with your types?

- `x = int(x)` casts `x` to an `int` (if possible)
- `x = float(x)` casts `x` to a `float` (if possible)
- `x = str(x)` casts `x` to a `str`

Type casting

Type casting

```
>>> x = 3.0
```

Type casting

```
>>> x = 3.0
>>> type(x)
<class "float">
```

Type casting

```
>>> x = 3.0
>>> type(x)
<class "float">
>>> x
3.0
```

Type casting

```
>>> x = 3.0
>>> type(x)
<class "float">
>>> x
3.0
>>> int(x)
3
```

Type casting

```
>>> x = 3.0
>>> type(x)
<class "float">
>>> x
3.0
>>> int(x)
3
>>> x = int(x)
```

Type casting

```
>>> x = 3.0
>>> type(x)
<class "float">
>>> x
3.0
>>> int(x)
3
>>> x = int(x)
>>> type(x)
<class "int">
```

Type Casting Rules

Type Casting Rules

- Can always cast between numeric types

Type Casting Rules

- Can always cast between numeric types
  - `float` $\rightarrow$ `int` rounds down `int(3.5)` $\rightarrow$ 3

Type Casting Rules

- Can always cast between numeric types
  - `float` $\rightarrow$ `int` rounds down `int(3.5)` $\rightarrow$ `3`
  - `int` $\rightarrow$ `float` adds `".0"` `float(3)` $\rightarrow$ `3.0`

Type Casting Rules

- Can always cast between numeric types
  - `float` → `int` rounds down `int(3.5)` → `3`
  - `int` → `float` adds `".0"` `float(3)` → `3.0`

- numeric → `str` always works

Type Casting Rules

- Can always cast between numeric types
  - `float` → `int` rounds down `int(3.5)` → `3`
  - `int` → `float` adds `".0"` `float(3)` → `3.0`

- numeric → `str` always works
  - `str(3)` → `"3"`

Type Casting Rules

- Can always cast between numeric types
  - `float` → `int` rounds down `int(3.5)` → `3`
  - `int` → `float` adds `".0"` `float(3)` → `3.0`
- numeric → `str` always works
  - `str(3)` → `"3"`
- `str` → numeric only works if the string in quotes would be a legal Python immediate type

Type Casting Rules

- Can always cast between numeric types
  - `float` → `int` rounds down `int(3.5)` → 3
  - `int` → `float` adds `".0"` `float(3)` → `3.0`
- numeric → `str` always works
  - `str(3)` → `"3"`
- `str` → numeric only works if the string in quotes would be a legal Python immediate type
  - `float("3.0")` → `3.0`

Type Casting Rules

- Can always cast between numeric types
  - `float` → `int` rounds down `int(3.5)` → 3
  - `int` → `float` adds `".0"` `float(3)` → `3.0`
- numeric → `str` always works
  - `str(3)` → `"3"`
- `str` → numeric only works if the string in quotes would be a legal Python immediate type
  - `float("3.0")` → `3.0`
  - `float("hello world")` → ValueError

Type Casting Rules

- Can always cast between numeric types
  - `float` → `int` rounds down `int(3.5)` → 3
  - `int` → `float` adds `".0"` `float(3)` → `3.0`
- numeric → `str` always works
  - `str(3)` → `"3"`
- `str` → numeric only works if the string in quotes would be a legal Python immediate type
  - `float("3.0")` → `3.0`
  - `float("hello world")` → ValueError
  - `int("3.0")` → ValueError

Type Casting Rules

- Can always cast between numeric types
  - `float` → `int` rounds down `int(3.5)` → 3
  - `int` → `float` adds `".0"` `float(3)` → `3.0`
- numeric → `str` always works
  - `str(3)` → `"3"`
- `str` → numeric only works if the string in quotes would be a legal Python immediate type
  - `float("3.0")` → `3.0`
  - `float("hello world")` → ValueError
  - `int("3.0")` → ValueError
  - `int(float("3.0"))` → `3`

Concept Check!

What will be the result of the following code blocks:

```
userInput = 1.5
answer = 1 + int(userInput)
print(answer)

userInput = "1.0"
answer = 2 * float(userInput)
print(answer)

userInput = "1.0"
answer = 2 * int(userInput)
print(answer)

num = 2 + 2
answer = "The answer is:  " + str(num)
print(answer)
```

Warning! Complicated Stuff Ahead!

Warning! Complicated Stuff Ahead!

(and you'll probably never need to use it)

Challenge

Question: Why can't we convert any string into a number?

Challenge

Question: Why can't we convert any string into a number?

What if we had to? What's the most reasonable way to do it?

Converting Strings by Unicode Encodings

Converting Strings by Unicode Encodings

- `ord(character)` converts a single character to `int` using it's Unicode encoding

Converting Strings by Unicode Encodings

- `ord(character)` converts a single character to `int` using it's Unicode encoding
- `chr(number)` converts an integer to `str` using by looking up in the Unicode table