Data Types

Harry Smith

Learning Objectives

- Understand what is meant by a "data type"
- List & use common operations including:
  - mathematical operations
  - relational operations
  - logical
- Recognize & debug common type errors
- Converting between values of different types & using `input()`

Computers & Data

Computers, Data, & Data Types

Computers are devices that store, retrieve, and manipulate data at extreme speeds.

Data Types allow us to understand how computers organize & use this data.

- **Data:** pieces of information
- **Data Type:** a category of information that defines a *set of possible values* that a member can have and the *set of operators* that can be used to manipulate those values.

Some Common Data Types

Data Type	Purpose	Sample Values	Sample Operations
`int`	whole (integer) numbers	`3`, `-14`, `0`	`+`, `-`, `*`, `/`
`float`	numbers with fractional parts	`3.0`, `-14.32`, `0.0`	`+`, `-`, `*`, `/`
`bool`	truth values	`True`, `False`	`and`, `or`, `not`
`str`	text	`"CIS 1100"`, `"False"`	`len()`, indexing & slicing
`None`	the absence of a value	`True`, `False`	`and`, `or`, `not`

Numeric Types

Numeric Types: `int`

`int` is a data type that represents whole **integer** numeric values.

- These values can be positive, negative, or zero
- No fractional (decimal) parts allowed
- e.g. `3`, `1`, `0`, `-10`, `-1033` are all examples of `int` values.
- Operations with `int` values are always precisely correct—no rounding error

Numeric Types: `float`

`float` is a data type that represents numbers that contain a fractional (decimal) part.

- These values can be positive, negative, or zero
- Can have a fractional part
- e.g. `3.0`, `1.4`, `0.0`, `-10.0`, `-1033.333`
- Operations with `float` values can have very small amounts of "precision" errors 😬

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

`int` vs. `float`

- **TL;DR:** mostly, they can be used interchangeably.
- Much of the arithmetic you do in Python converts `int` values to `float` values automatically. (More in a minute.)
- `int` values are *enumerable*, which is nice for picking options out of a sequence.

Numeric Types: Operations

For numeric types like `int` and `float`, the important operators are all mathematical.

Operator	Operation	Example with `int` values	Output Value	Output Type
`+`	Addition	`3 + 5`	`8`	`int`
`-`	Subtraction	`4 - 6`	`-2`	`int`
`*`	Multiplication	`2 * 3`	`6`	`int`
`/`	Division	`3 / 2`	`1.5`	`float`

Numeric Types: Operations

Notice how `float` is "contagious:" when a part of the expression is a `float`, the output will be a `float`.

Operator	Operation	Example with `int` and `float` values	Output Value	Output Type
`+`	Addition	`3.1 + 5`	`8.1`	`float`
`-`	Subtraction	`4.0 - 0.86`	`3.14`	`float`
`*`	Multiplication	`-2.0 * 3`	`-6.0`	`float`
`/`	Division	`3.0 / 2.0`	`1.5`	`float`

Integer Division & Modulo

Operator	Operation	Example with `int` values	Output Value	Output Type
`//`	Integer Division	`5 // 2`	`2`	⚠️`int`⚠️
`%`	Modulo (or "mod")	`5 % 2`	`1`	`int`

Integer Division

- Allows us to divide two `int` values and get an `int` as a result.
- Do regular division arithmetic, and then **truncate** the result by removing the fractional part
  - Whereas `3 / 2` has a value of `1.5`, we know that `3 // 2` has a value of `1`
  - Whereas `4 / 2` has a value of `2.0`, we know that `4 // 2` has a value of `2`

Modulo

- `a % b` calculates the remainder left after dividing `a` by `b` with integer division.
- Example: `16 % 5` evaluates to `1`—why?
  - `5` "goes into" `16` three times (i.e. `16 // 5` evaluates to `3`)
  - If we calculate `5 * 3`, we get `15` as a result.
  - The remainder between our answer and the right one is `16 - 15`, or `1`.
  - *"If we divide 16 slices of pizza among 5 people, how many slices will be left over?"*

Some Modulo Patterns

Example Expression	Example Result
`0 % 3`	`0`
`1 % 3`	`1`
`2 % 3`	`2`
`3 % 3`	`0`
`4 % 3`	`1`
`5 % 3`	`2`

Some Modulo Patterns

Example Expression	Example Result
`12 % 1`	`0`
`12 % 2`	`0`
`12 % 3`	`0`
`12 % 4`	`0`
`12 % 5`	`2`
`12 % 6`	`0`
`12 % 7`	`5`

Properties of Modulo

- The output of `a % b` is always a number between `0` and `b - 1`.
- If `a` is evenly divisible by `b`, then `a % b` will always output `0`.
- A general identity: $a = (a//b) * b + (a \% b)$

Booleans & Logical Operators

Booleans & `bool`

- Programming isn't just numbers—also have notions of truth and logic.
- Computers use **boolean logic**: a logic system with just two values.
- The `bool` data type consists of **just two** values: `True` and `False`.
  - They're words, but they're not treated as text ➡️ no quotes!
  - Make sure to spell them with capital letters.

Logical Operators

- Logical systems give us ways of writing expressions that include variables.
  - Evaluating whether a expressions is `True` or `False` depends on the values of the variables
- We can create complex & interesting expressions using **logical operators**
  - These are *conjunction, disjunction, & negation*
  - Or, more commonly: `and`, `or`, & `not`.'

`and`

Expressions with `and` evaluate to `True` only when both operands are both `True`.

`a`	`b`	`a and b`
`True`	`True`	`True`
`True`	`False`	`False`
`False`	`True`	`False`
`False`	`False`	`False`

`or`

Expressions with `or` evaluate to `True` as long as one operand is `True`.

`a`	`b`	`a or b`
`True`	`True`	`True`
`True`	`False`	`True`
`False`	`True`	`True`
`False`	`False`	`False`

`not`

`not` flips the value of the expression it's applied to.

- `not` is an example of a *unary operator*: works on a single expression.

`a`	`not a`
`True`	`False`
`False`	`True`

Finding the Truth of the Matter

```
raining = True
windy = False
not (raining and windy) or not raining and not windy
```

- Have to replace variables with their values to get an answer.
- Parentheses evaluated first.
- `not` comes before `and` comes before `or`.

Finding the Truth of the Matter

```
raining = True
windy = False
not (raining and windy) or not raining and not windy
not (True and False) or not True and not False
not (False) or not True and not False
True or not True and not False
True or False and not False
True or False and True
True or False
True
```

Strings

Strings

- Used to represent text
- `str` is the name of the type
  - Its values can be any sequence of valid **characters** (letters, digits, punctuation, or spacing)
  - Literals are denoted using pairs of quotation marks (can use `"`, `'`)
- Examples of `str` values:
  - `"Harry S. Smith"`
  - `"3330 Walnut Street"`
  - `"!@#$%^&*()0123456789"`

Strings & Length

It often makes sense to discuss the length of a `str` value, or the number of characters it contains.

`str`	Length	`str`	Length
"Harry"	5	"👀"	1
"HarrySmith"	10	" "	1
"Harry Smith"	11	""	0
"1100?"	5	"!@#$"	4

Finding the Length of a String

The expression `len(s)` evaluates to the `int` representing the number of characters in `s`.

```
long_word = "antidisestablishmentarianism"
length = len(long_word)
print(length)
```

🖨️👇

```
28
```

Combining Strings

The **concatenation** operation is the process of joining two strings together end-to-end.

- The operator is `+`, but it's not addition!
- Two `str` values are glued together with nothing added between them.

```
>>> "CIS" + "1100"
"CIS1100"
>>> "Grace" + "Hopper"
"GraceHopper"
>>> "CIS" + 1100
TypeError: can only concatenate str (not "int") to str
>>> "1" + "1"
"11"
```

Repeating Strings

Less commonly useful, but still of note: the string duplication operator (`*`) repeats a `str` value a number of times.

```
>>> "ha" * 1
"ha"
>>> "ha" * 2
"haha"
>>> "ha" * 4
"hahahaha"
>>> "ha" * 10
"hahahahahahahahahaha"
```

`None` : The "Nothing to See Here" Type

`None` is a type with only a single value: `None`

- Used to signify the *absence* of a value in many situations.
- Sometimes we write expressions that don't have a meaningful value, so: `None`.

```
result = print("Hello, world!")
print(result)
```

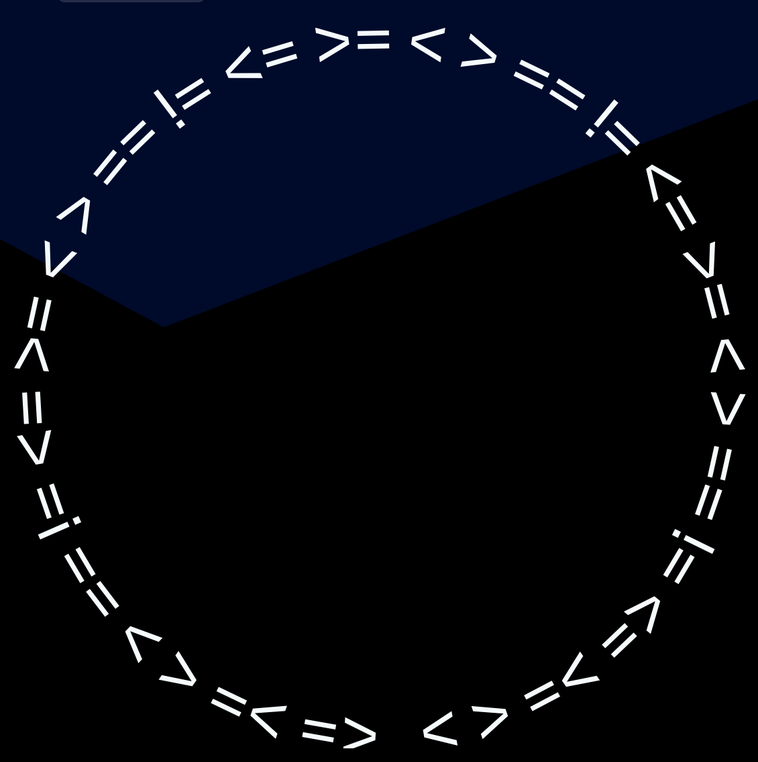
🖨️👇

```
None
```

Relational Operators

Relational Operators

- Group of operators that can be applied to values of different data types
- Provide us ways of comparing two values for order or equality.
- The output data type is always a `bool`.



Equality (`==`)

The `==` ("double equals") operator, allows us to ask if two values are equivalent to each other.

Expression	Result
`4 == 4`	`True`
`4.0 == 4`	`True`
`"4" == 4`	`False`

Not Equality (`!=`)

The `!=` ("not equals") operator, allows us to ask if two values are different from each other.

Expression	Result
`4 != 4`	`False`
`5 != 4`	`True`
`"Comp" != "Sci"`	`True`

Ordering (`<`, `<=`, `>`, `>=`)

Evaluate the relative ordering of two values, producing a `bool`.

- The comparison operators must take in two values of the same kind: both numeric (`int` or `float`), both `str`, or both `bool`

Expression	Result
`4 > 5`	`False`
`9 <= 9`	`True`
`"carrot" > "banana"`	`True`
`4 > "howdy"`	🚨Error! Type mismatch. 🚨

⛓️ Chained Ordering ⛓️

Convenient and succinct way of determining whether or not a value fits within a certain range.

- `10 >= 0 > -10` is the same as `10 >= 0 and 0 > -10`

Examples
`0 < x <= 20`
`"zebra" > my_animal > "elephant"`

leap_year.py

Example: Leap Years

Let's write code that determines whether or not a year counts as a Leap Year. From Wikipedia:

> A leap year [...] is a calendar year that contains an additional day [...] compared to a common year. The 366th day [...] is added to keep the calendar year synchronised with the astronomical year or seasonal year.

Example: Leap Years

A year is a Leap Year if:

- The year number is divisible by four and the year number is not divisible by 100, or
- The year number is divisible by 400

Example: Leap Years

Our toolkit:

- **Divisibility**: `a % b == 0` when `a` is divisible by `b`
- **Logical Operators**: `and` & `or` can be used to combine multiple boolean expressions

Example: Leap Years

A year is a Leap Year if:

- The year number is divisible by four and the year number is not divisible by 100, or
- The year number is divisible by 400

A year is a Leap Year if:

- The `year % 4 == 0` and `year 5 100 != 0`, or
- The `year % 400 == 0`

Example: Leap Years

A year is a Leap Year if:

- `(year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)`

Example: Refactoring Leap Year

leap_year.py

```
year = 2024
is_leap_year = (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
print(f"Is {year} a leap year? {is_leap_year}")
```