DATA TYPES

DATA TYPES

In this section we'll introduce Python **data types**, review their properties, and cover how to identify and convert between them

TOPICS WE'LL COVER:

- Data Types
- The Type Function
- Type Conversion
- Iterables
- Mutability

GOALS FOR THIS SECTION:

- Review the basics of Python data types
- Learn how to determine an object's data type
- Learn to convert between compatible data types
- Understand the concepts of iterability & mutability

PYTHON DATA TYPES

- Data Types
- The Type Function
- Type Conversion
- Iterables
- Mutability

Python has a series of built-in **data types** with different properties and use cases

These can be grouped into the following categories:

Numeric

Text

Boolean

None

Sequence

Mapping

Set

PYTHON DATA TYPES

- Data Types
- The Type Function
- Type Conversion
- Iterables
- Mutability

Single value *(simple)*:

Numeric

Represents numeric values

- Integer(**`int`**) – `5`
- Float(**`float`**) – `5.24`
- Complex(**`complex`**) – `5 + 2j`

Text

Represents sequences of characters, usually text

- String(**`str`**) – `'snowboard'`

Boolean

Represents True and False values

- Boolean(**`bool`**) – `True`, `False`

None

Represents the absence of a value

- **`NoneType`** – `None`

Multiple values:

Sequence

Represents sequences of values, usually text or numeric

- List(**`list`**) – `[1, 3, 5, 7, 9]`
- Tuple(**`tuple`**) – `('snowboard', 'skis')`
- Range(**`range`**) – `range(1, 10, 2)`

Mapping

Maps keys to values for efficient information retrieval

- Dictionary(**`dict`**) – `{'snowboard': 24, 'skis': 17}`

Set

Represents a collection of unique, non-duplicate values

- Set(**`set`**) – `{'snowboard', 'skis'}`
- Frozen Set(**`frozenset`**) – `{'snowboard', 'skis'}`

THE TYPE FUNCTION

- Data Types
- The Type Function
- Type Conversion
- Iterables
- Mutability

The **type()** function will return the data type of the object passed to it

```
type(1)
```

```
int
```

```
type('snowboard')
```

```
str
```

```
type(True)
```

```
bool
```

```
type(None)
```

```
NoneType
```

```
type([1, 3, 5, 7])
```

```
list
```

```
type({'a': 3, 'b': 5})
```

```
dict
```

PRO TIP: Use type() if you are getting a TypeError or unexpected behavior when passing data through a function to make sure the data type is correct; it's not uncommon for values to be stored incorrectly

TYPE CONVERSION

- Data Types
- The Type Function
- Type Conversion
- Iterables
- Mutability

Convert data types by using "<data type>(object)"

EXAMPLE *Converting data into an integer data type*

```
int('123')
```

```
123
```

*Using **int** converts the text string of '123' into an integer data type*

Errors in Python will **cause the code to stop running**, so it's important to learn to diagnose and fix them

```
int([1, 2, 3])
```

```
TypeError
```

*Using **int** attempts to convert the list into an integer data type, but returns a **TypeError** instead*

This operation isn't valid because the list has multiple values, while an integer can only be one value

```
TypeError: int() argument must be a string, a bytes-like object or a number, not 'list'
```

ITERABLES

- Data Types
- The Type Function
- Type Conversion
- Iterables
- Mutability

Iterables are data types that can be iterated, or looped through, allowing you to move from one value to the next

These data types are considered iterable:

Sequence

Mapping

Set

Text

While text strings are treated as a single value, individual characters in a text string can be iterated through

MUTABILITY

- Data Types
- The Type Function
- Type Conversion
- Iterables
- Mutability

A data type is **mutable** if it can be modified after its creation

Mutable Data Types

Flexible – can add, remove, change values in the object

- Lists
- Dictionaries
- Sets

Immutable Data Types

To modify a value must delete and recreate the entire object

- Integers
- Floats
- Strings
- Booleans
- Tuples
- Frozenset

Note that all simple data types are immutable

KEY TAKEAWAYS

Python has a wide variety of **data types** with different properties

- *It's important to understand them at a high level for now, as we'll dive deeper into each later in the course*

The **type() function** allows you to determine an object's data type

- *You can convert between data types if the underlying values are compatible*

Iterables are data types with values that you can loop through

- *Text strings are iterable despite containing a single value, as you can iterate through its characters*

Data types can be **mutable** or **immutable**

- *Mutable data types can be modified after their creation, while immutable data types cannot be changed without being overwritten*

VARIABLES

VARIABLES

In this section we'll introduce the concept of **variables**, including how to properly name, overwrite, delete, and keep track of them

TOPICS WE'LL COVER:

- Variable Assignment
- Overwriting & Deleting
- Naming Conventions
- Tracking Variables

GOALS FOR THIS SECTION:

- Learn to assign variables in Python
- Understand the behavior for overwriting variables
- Learn the rules & best practices for naming variables

VARIABLE ASSIGNMENT

- Variable Assignment
- Overwriting & Deleting
- Naming Conventions
- Tracking Variables

Variables are containers used to label and store values for future reference

- They can store any Python data type (and even calls to functions!)

```
variable_name = value
```

Intuitive label assigned to the variable

Examples:

- *price*
- *city*
- *heights*

Initial value assigned to the variable

Examples:

- *10.99*
- *'Los Angeles'*
- *[180, 173, 191]*

Make sure you add the **space** on both sides of the equal sign!

VARIABLE ASSIGNMENT

- Variable Assignment
- Overwriting & Deleting
- Naming Conventions
- Tracking Variables

Variables are containers used to label and store values for future reference

- They can store any Python data type (and even calls to functions!)

EXAMPLE *Creating a price variable*

```
price = 5

print(price)
```

```
5
```

*We're creating a variable named **price** and assigning it a value of 5, then we're printing the variable, returning its value*

PRO TIP: Only assign variables for values that can change or will be used repeatedly; if you're not sure, it's likely a good idea to assign a variable

```
price = 5

print(price + 1)
```

```
6
```

Any operation that can be performed on the value of a variable can be performed using the variable name

*Here we're printing the result of adding 1, a hard coded value, to the **price***

VARIABLE ASSIGNMENT

- Variable Assignment
- Overwriting & Deleting
- Naming Conventions
- Tracking Variables

Variables are containers used to label and store values for future reference

- They can store any Python data type (and even calls to functions!)

EXAMPLE | *Creating a price list variable*

```
price_list = [2.50, 4.99, 10, None, 'PROMO']

print_price_list = print(price_list)

print_price_list
```

```
[2.5, 4.99, 10, None, 'PROMO']
```

*First, we're creating a variable named **price_list** and assigning it to list of values*

*Then we're assigning a call to the print function with our **price_list** variable as input*

Note that assigning the print() function to a variable here doesn't necessarily improve our program over simply calling print() outside of it, but it's worth knowing **even a function call can be assigned to a variable**

ASSIGNMENT: VARIABLE ASSIGNMENT

NEW MESSAGE

February 3, 2022

From: **Sally Snow** *(Ski Shop Manager)*

Subject: **Tax Calculation**

Hi there, welcome to the Maven Ski Company!

For your first task, I'd like you to change the code that adds a dollar in tax to our price before printing.

The tax will no longer be fixed at one dollar, so:

- Create a 'tax' variable and assign it a value of 2
- Create a 'total' variable equal to 'price' + 'tax'
- Print 'total'

Thanks in advance!

tax_code.ipynb

Reply | Forward

Results Preview

```
print(total)
```

```
7
```

OVERWRITING VARIABLES

- Variable Assignment
- Overwriting & Deleting
- Naming Conventions
- Tracking Variables

You can **overwrite a variable** by assigning a new value to it

- They can be overwritten any number of times

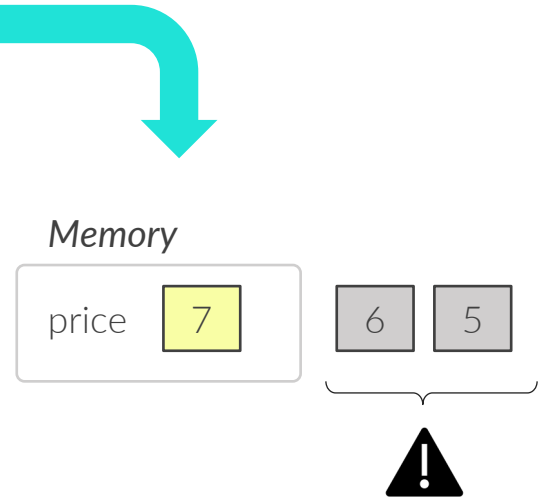
```
price = 5
price = 6
price = 7

print(price)
```

```
7
```

*Python stores the value of 5 in memory when it is assigned to **price***

Then price stores the value 6, and 5 gets removed from memory

Then price stores the value 7, and 6 gets removed from memory



When you overwrite a variable, **its previous value will be lost** and cannot be retrieved

```
price = 5
new_price = 6

print(price, new_price)
```

```
5 6
```

Consider creating a new variable for a new value rather than overwriting

OVERWRITING VARIABLES

You can **overwrite a variable** by assigning a new value to it

- They can be overwritten any number of times

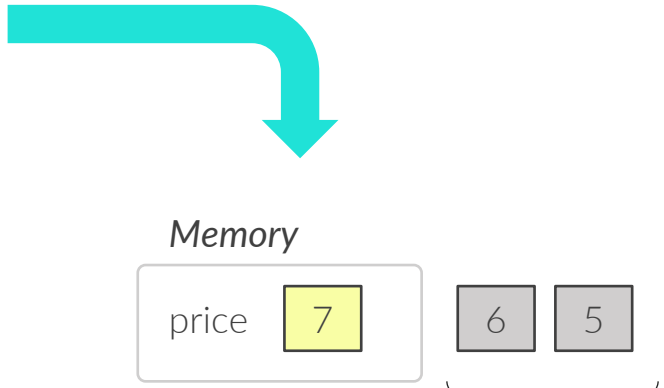
```
price = 5
price = 6
price = 7

print(price)
```

```
7
```

*Python stores the value of 5 in memory when it is assigned to **price***

Then price stores the value 6, and 5 gets removed from memory

Then price stores the value 7, and 6 gets removed from memory




```
old_price = 5
price = 6

print(old_price, price)
```

```
5 6
```

Or create a new variable for the old value

When you overwrite a variable, **its previous value will be lost** and cannot be retrieved

OVERWRITING VARIABLES

- Variable Assignment
- Overwriting & Deleting
- Naming Conventions
- Tracking Variables

You can **overwrite a variable** by assigning a new value to it

- They can be overwritten any number of times

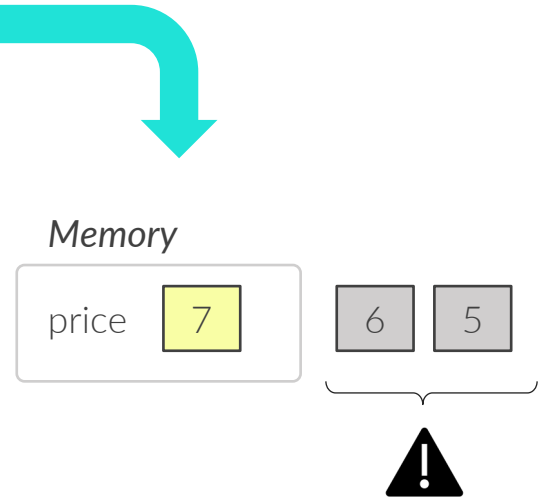
```
price = 5
price = 6
price = 7

print(price)
```

```
7
```

*Python stores the value of 5 in memory when it is assigned to **price***

Then price stores the value 6, and 5 gets removed from memory

Then price stores the value 7, and 6 gets removed from memory



When you overwrite a variable, **its previous value will be lost** and cannot be retrieved

PRO TIP: Use variables like 'new_price' and 'old_price' when testing programs with different values to make sure you don't lose important data – once the code works as needed, remove any extra variables!

OVERWRITING VARIABLES

Variables can also be **assigned as values to other variables**

- The underlying value is assigned, but there is no remaining association between the variables

```
price = 5
new_price = 6
price = new_price
new_price = 7

print(price)
```

```
6
```

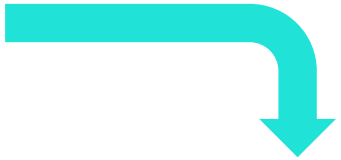
*Python stores the value of 5 in memory when it is assigned to **price***

*Then stores the value of 6 when it is assigned to **new_price***

*Then assigns the value of **new_price**, which is 6, to **price,** 5 is no longer assigned to a variable, so gets removed*

*Then assigns the value of 7 to **new_price***

*Note that **price** is still equal to 6, it's not tied to the value of **new_price**!*



Memory

price	6
new_price	7

5

DELETING VARIABLES

- Variable Assignment
- Overwriting & Deleting
- Naming Conventions
- Tracking Variables

The **del** keyword will permanently remove variables and other objects

```
price = 5
del price

print(price)
```

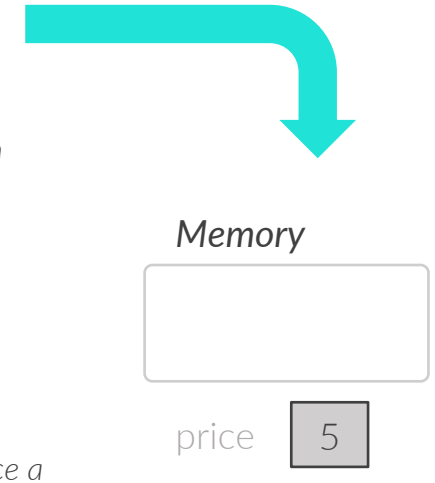
```
---------------------------------------
---------------------------------------
NameError
Traceback (most recent call last)
/var/folders/f8/075hbnj13wb0f9yzh9k4ny
z00000gn/T/ipykernel_36120/432900761.p
y in <module>
      2 del price
      3
----> 4 print(price)

NameError: name 'price' is not defined
```

*Python stores the value of 5 in memory when it is assigned to **price***

*Then **del** removes it from memory*

*When we try to print **price**, we get an **Error**, as the variable no longer exists*

NameError occurs when we reference a variable or object name that isn't defined



PRO TIP: Deleting objects is generally unnecessary, and mostly used for large objects (like datasets with 10k+ rows); in most cases, reassign the variables instead and Python will get rid of the old value!

NAMING RULES

- Variable Assignment
- Overwriting & Deleting
- Naming Conventions
- Tracking Variables

Variables have some basic **naming rules**

Variable names **can**:

- Contain letters (case sensitive!)
- Contain numbers
- Contain underscores
- Begin with a letter or underscore

Variable names **cannot**:

- Begin with a number
- Contain spaces or other special characters *(*, &, ^, -, etc.)*
- Be reserved Python keywords like **del** or **list**

PRO TIP: "Snake case" is the recommended naming style for Python variables, which is all lowercase with words separated by underscores (first_second_third, new_price, etc.)

NAMING RULES

- Variable Assignment
- Overwriting & Deleting
- Naming Conventions
- Tracking Variables

Variables have some basic **naming rules**

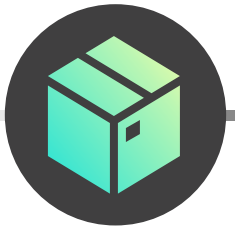
Valid variable names:

- price_list_2019
- _price_list_2019
- PRICE_LIST_2019
- pl2019

Invalid variable names

- 2019_price_list *(starts with a number)*
- price_list-2019 *(has special characters)*
- 2019 price list *(has spaces)*
- list *(reserved Python keyword)*

PRO TIP: Give your variables intuitive names to make understanding your code easier – instead of PL19, consider something like price_list_2019 or prices_2019



TRACKING VARIABLES

Use “%who” and “%whos” to **track the variables** you’ve created

- Variable Assignment
- Overwriting & Deleting
- Naming Conventions
- Tracking Variables

```
price = 10
product = 'Super Snowboard'
Date = '10-Jan-2021'
dimensions = [160, 25, 2]
```

```
%who
```

```
Date     dimensions      price   product
```

%who *returns variable names*

```
%whos
```

```
Variable     Type    Data/Info
-----------------------------
Date         str     10-Jan-2021
dimensions   list    n=3
price        int     10
product      str     Super Snowboard
```

%whos *returns variable names, types, and information on the data contained*

Magic commands that start with “%” only work in the iPython environments, which applies to Jupyter and Colab

ASSIGNMENT: FIXING VARIABLES

NEW MESSAGE

February 3, 2022

From: **Sally Snow** *(Ski Shop Manager)*

Subject: **Price List Creation**

Hi there, thanks for you help on the tax code!

The file attached contains the attempts of our previous intern to create variables representing our 2018, 2019, and 2020 price lists, among others.

Please make sure these variables are given names in the format 'price_list_year' by filling in the correct year and cleaning up any unnecessary variable names.

Thanks in advance!

price_lists.ipynb

Reply Forward

Results Preview

```
Variable           Type    Data/Info
-----------------------------------
price_list_2018    list    n=5
price_list_2019    list    n=5
price_list_2020    list    n=5
```

KEY TAKEAWAYS

Variables are containers used to **store values** from any data type

- *These values are stored in memory and can be referenced repeatedly in your code*

Overwrite variables by assigning new values to them

- *The old value will be lost unless it's assigned to another variable*

Variable names must follow Python's **naming rules**

- *"Snake case" is the recommended naming style (all lowercase with underscores separating each word)*

Give variables **intuitive** names

- *Even though you can use magic commands to track variables, good names save a lot of time & confusion*

NUMERIC DATA

NUMERIC DATA

In this section we'll cover **numeric data types**, including how to convert between them, perform arithmetic operations, and apply numeric functions

TOPICS WE'LL COVER:

- Numeric Data Types
- Numeric Type Conversion
- Arithmetic Operators
- Numeric Functions

GOALS FOR THIS SECTION:

- Review the different numeric data types
- Learn to convert between data types
- Perform moderately complex arithmetic operations

NUMERIC DATA TYPES

Numeric Data Types

Numeric Type Conversion

Arithmetic Operators

Numeric Functions

There are 3 types of **numeric data types** in Python:

1. **Integers (int)** – whole numbers without decimal points
   - *Examples:* `3`, `42`, `-14`
2. **Floats (float)** – real numbers with decimal points
   - *Examples:* `3.14`, `-1.20134`, `0.088`
3. **Complex (complex)** – numbers with real and imaginary portions
   - *Examples:* `3 + 2j`, `2.4 - 1j`, `-2j`

Never worked with complex numbers before? Don't worry! They are rarely used in **data analytics**

NUMERIC TYPE CONVERSION

- Numeric Data Types
- Numeric Type Conversion
- Arithmetic Operators
- Numeric Functions

Use "<data type>(object)" to **convert any number** into a numeric data type

You can convert floats into integers:

```
number = 1.9999
type(number)
```

```
float
```

```
fixed_number = int(number)
fixed_number, type(fixed_number)
```

```
(1, int)
```

Note that converting to int ***rounds down***

Integers into floats:

```
number = 1
fixed_number = float(number)
fixed_number, type(fixed_number)
```

```
(1.0, float)
```

And even strings into floats (or integers):

```
text = '1.2'
type(text)
```

```
str
```

```
fixed_number = float(text)
fixed_number, type(fixed_number)
```

```
(1.2, float)
```

As long as they only contain numeric characters:

```
number = 'abc123'
fixed_number = int(number)
```

Note that this string ***has letters***

```
ValueError: invalid literal for int() with base 10: 'abc123'
```

ValueError *occurs when a function receives an input with an inappropriate value*

ARITHMETIC OPERATORS

- Numeric Data Types
- Numeric Type Conversion
- Arithmetic Operators
- Numeric Functions

`+`	**Addition**	Adds two values	`5 + 3 = 8`
`-`	**Subtraction**	Subtracts one value from another	`5 - 3 = 2`
`*`	**Multiplication**	Multiplies two values	`5 * 3 = 15`
`/`	**Division**	Divides one value by another	`5 / 3 = 1.6666666667`
`//`	**Floor Division**	Divides one value by another, then rounds down to the nearest integer	`5 // 3 = 1`
`%`	**Modulo**	Returns the remainder of a division	`5 % 3 = 2`
`**`	**Exponentiation**	Raises one value to the power of another	`5 ** 3 = 125`

ORDER OF OPERATIONS

- Numeric Data Types
- Numeric Type Conversion
- Arithmetic Operators
- Numeric Functions

Python uses the standard PEMDAS **order of operations** to perform calculations

1. **P**arentheses
2. **E**xponentiation
3. **M**ultiplication & **D**ivision (including Floor Division & Modulo), from left to right
4. **A**ddition & **S**ubtraction, from left to right

Without Parentheses

```
3*6 + 5 - 2**3 - 1
```

Exponentiation first

```
3*6 + 5 - 8 - 1
```

Then multiplication

```
18 + 5 - 8 - 1
```

Finally, addition & subtraction

```
14
```

With Parentheses *(to control execution)*

```
3*((6 + 5) - 2**(3 - 1))
```

Addition & subtraction in inner parentheses first

```
3 * (11 - 2**2)
```

Then exponentiation in outer parenthesis

```
3 * (11 - 4)
```

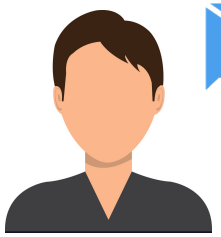
Then subtraction in outer parenthesis

```
3 * 7
```

Finally, multiplication

```
21
```

ASSIGNMENT: ARITHMETIC OPERATORS



NEW MESSAGE

February 4, 2022

From: **Ricardo Nieva** *(Financial Planner)*

Subject: **Financial Calculations**

Hi there!

Can you help calculate the following numbers?

- Gross profit from selling a snowboard
- Gross margin from selling a snowboard
- Price needed for a gross margin of 70%
- Sales tax on a snowboard sale
- Amount of money if the gross profit from selling 5 snowboards is invested for one year

There are more details in the file attached.

financial_calculations.ipynb

Reply Forward

Results Preview

```
print(gross_profit)
print(gross_margin)
print(price_needed_for70)
print(sales_tax)
print(amount_after_1yr)
```

```
300.0
0.6000120002400048
666.6333333333332
39
1575.0
```

NUMERIC FUNCTIONS

- Numeric Data Types
- Numeric Type Conversion
- Arithmetic Operators
- Numeric Functions

Single number:

round	*Rounds a number to a specified number of digits*	`round(number, # of digits)`
abs	*Returns the absolute value of a number*	`abs(number)`

Multiple numbers:

sum	*Sums all numbers in an iterable*	`sum(iterable)`
min	*Returns the smallest value in an iterable*	`min(iterable)`
max	*Returns the largest value in an iterable*	`max(iterable)`

ROUND & ABS

- Numeric Data Types
- Numeric Type Conversion
- Arithmetic Operators
- Numeric Functions

The **round()** function rounds a number to a specified number of digits

The **abs()** function returns the absolute value of a number

```
round(3.1415926535, 2)
```

This rounds the number to 2 decimal places

```
3.14
```

```
abs(-3)
```

This will always return a positive number

```
3
```

```
round(3.1415926535)
```

If number of digits isn't supplied, it will round to the nearest integer

```
3
```

```
round(3.51)
```

It rounds down decimals < 0.5 and rounds up decimals >= 0.5

```
4
```

SUM, MIN & MAX

- Numeric Data Types
- Numeric Type Conversion
- Arithmetic Operators
- Numeric Functions

The **sum()**, **min()**, and **min()** functions perform sum, minimum, and maximum calculations on iterable data types that only contain numeric values

```
sum((3, 4, 5))
```

This sums the numbers in the tuple

```
12
```

```
min([3, 4, 5])
```

This returns the smallest value from the list

```
3
```

```
max({3, 4, 5})
```

This returns the largest value from the set

```
5
```

ASSIGNMENT: NUMERIC FUNCTIONS

NEW MESSAGE

February 4, 2022

From: **Sally Snow** *(Ski Shop Manager)*

Subject: **Customer Questions**

Hi there!

Could you quickly run these numbers for me?

- What is the highest priced item we sell?
- What is the lowest priced item we sell?
- How much would it cost for a customer to purchase two of every item, rounded to the nearest dollar?

Thanks in advance! (the price list is attached)

customer_questions.ipynb

Reply Forward

Results Preview

```
price_list = [129.99, 99.99, 119.19, 99.99, 89.99, 79.99, 49.99]
```

```
[redacted]

print(lowest_price, highest_price)
```

```
49.99 129.99
```

```
[redacted]
```

```
1338
```

KEY TAKEAWAYS

Data analysts typically work with **integer** & **float** numeric data types

- *Complex numbers are the third numeric data type, but it's unlikely you'll ever use them*

Arithmetic operations follow the **PEMDAS** order of operations

- *Use parentheses to control the order in which the operations are performed*

Python has **built-in functions** that work with specifically with numbers

- *Round, abs, sum, min, and max help perform simple operations that you'd otherwise have to code yourself*

STRINGS

STRINGS

In this section we'll review the fundamentals of **strings**, a date type used to store text, and cover functions and methods to manipulate them

TOPICS WE'LL COVER:

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

GOALS FOR THIS SECTION:

- Create strings from scratch or from other data types
- Learn to manipulate strings using arithmetic, indexing, slicing, and string methods
- Use f-strings to incorporate variables into strings and make them dynamic

STRING BASICS

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

Strings are sequences of character data commonly used to store text

You can create a string by using quotation marks, or converting from numbers:

Single *quotes*

```
new_string = 'This is a "string".'
print(new_string)
```

```
This is a "string".
```

Double *quotes*

```
newer_string = "Double quotes allow us to use ' inside."
print(newer_string)
```

```
Double quotes allow us to use ' inside.
```

Triple *quotes*

```
# triple quotes
newerer_string = '''Triple quotes allow us to
write multiline strings. We also don't
have to worry about "errors" due to using
single or double quotes. Pretty cool!
'''
print(newerer_string)
```

```
Triple quotes allow us to
write multiline strings. We also don't
have to worry about "errors" due to using
single or double quotes. Pretty cool!
```

Convert *to string*

```
number = 22.5
string = str(number)

string
```

```
'22.5'
```

The quotes indicate this is a string

```
print(string)
```

```
22.5
```

Using print() cleans up the quotes

STRING ARITHMETIC

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

Some **arithmetic operators** are compatible with strings

You can **concatenate** strings using +

```
'Snow' + 'board'
```

```
'Snowboard'
```

```
subject = 'My'
noun = 'Maven snowboard'
verb = 'cost'
price = 22.55

subject + ' ' + noun + ' ' + verb + ' $' + str(price) +'!'
```

```
'My Maven snowboard cost $22.55!'
```

You can concatenate any number of strings

The addition of ' ' to add spaces, as well as adding punctuation like '!' or '$' is common

You can **repeat** strings using *

```
'Repeat 3 times: ' + ("I will not steal my coworker's pen! ") * 3
```

```
"Repeat 3 times: I will not steal my coworker's pen! I will not steal
my coworker's pen! I will not steal my coworker's pen! "
```

Repeating strings is not very common in practice

STRING INDEXING

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

Strings, along with sequence data types, can be navigated via their **index**

- Python is **0-indexed**, which means that the first item in a sequence has an index of 0, not 1

```
message = 'I hope it snows tonight!'
```

message

0	I
1	
2	h
3	o
4	p
…	
22	t
23	!

```
message[0]
```

```
'I'
```

```
message[1]
```

```
' '
```

```
message[2]
```

```
'h'
```

String[index] *returns a specified character:*

- *String[0] returns the first character*
- *String[1] returns the second character*
- *String[2] returns the third character*

Indexing is a critical concept to master for data analysis, and while the 0-index can be confusing at first, with practice it will be second nature

STRING INDEXING

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

Strings, along with sequence data types, can be navigated via their **index**

- Python is **0-indexed**, which means that the first item in a sequence has an index of 0, not 1

```
message = 'I hope it snows tonight!'
```

```
message[-1]
'!'
```

String[index] *returns a specified character:*

- *String[-1] returns the last character*

message

```
-24   I
-23
...
-5    i
-4    g
-3    h
-2    t
-1    !
```

*You can also access individual characters in a text string by specifying their **negative** index*

Negative indices begin at the end of the object and work backwards

Since there is no negative 0, this starts at -1

⚠ Indexing a value **greater than the length of the string or object** results in an **IndexError**

```
message[52]
```

```
IndexError: string index out of range
```

ASSIGNMENT: STRING INDEXING

NEW MESSAGE

February 5, 2022

From: **Sierra Schnee** *(Security Consultant)*

Subject: **Password Retrieval**

Hi there!

I need your help deciphering a hidden message. Our previous intern created a complex password storage procedure.

I've determined one of his passwords is 'Maven' , but what I'm interested in is the index of the letters used to spell 'Maven' in the attached text messages, as this will help unlock new passwords.

Thanks in advance!

suspicious_texts.ipynb

Reply Forward

Results Preview

```
text1 = '          Your friend Mark'
text2 = 'was'
text3 = 'having'
text4 = 'a great day'
text5 = 'on the mountain'
```

```
maven = text1[10] + text2[1] + text3[1] + text4[2] + text5[4]

print('The secret password is ' + maven)
```

```
The secret password is Maven
```

STRING SLICING

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

Slicing returns multiple elements in a string, or sequence, via indexing

```
[start:stop:step size]
```

Index value to start with
(default is 0)

Index value to stop at, not inclusive
(default is all the values after the start)

Amount to increment each subsequent index
(default is 1)

As with indexing, **slicing is a critical concept to master for data analysis**, as it allows you to manipulate a wide variety of data formats

STRING SLICING

EXAMPLE | *Grabbing components of a text string*

```
message = 'I hope it snows tonight!'
message[0:6:1]
```

```
'I hope'
```

How does this code work?

- **Start: 0** – start with the first element in the string
- **Stop: 6** – stop at the seventh element (index of 6) in the string without including it
- **Step size: 1** – return every single element in the range

STRING SLICING

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

EXAMPLE | *Grabbing components of a text string*

```
message = 'I hope it snows tonight!'
message[:6]
```

```
'I hope'
```

How does this code work?

- **Start: 0** – start with the first element in the string by default
- **Stop: 6** – stop at the seventh element (index of 6) in the string without including it
- **Step size: 1** – return every single element in the range by default

STRING SLICING

EXAMPLE | *Grabbing components of a text string*

```
message = 'I hope it snows tonight!'
message[2:6]
```

```
'hope'
```

How does this code work?

- **Start: 2** – start with the third element (index of 2) in the string
- **Stop: 6** – stop at the seventh element (index of 6) in the string without including it
- **Step size: 1** – return every single element in the range by default

STRING SLICING

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

EXAMPLE | *Grabbing components of a text string*

```
message = 'I hope it snows tonight!'
message[6:]
```

```
' it snows tonight!'
```

How does this code work?

- **Start: 6** – start with the seventh element (index of 6) in the string
- **Stop: 25** – stop at the end of the string
- **Step size: 1** – return every single element in the range by default

STRING SLICING

EXAMPLE | *Grabbing components of a text string*

```
message = 'I hope it snows tonight!'
message[0:6:2]
```

```
'Ihp'
```

How does this code work?

- **Start: 0** – start with the first element in the string
- **Stop: 6** – stop at the seventh element (index of 6) in the string without including it
- **Step size: 2** – return every other element in the range

STRING SLICING

EXAMPLE | *Grabbing components of a text string*

```
message = 'I hope it snows tonight!'
message[::-1]
```

```
'!thginot swons ti epoh I'
```

How does this code work?

- **Start: -1** – start with the last element in the string by default, due to the negative step size
- **Stop: -25** – stop at the beginning of the string
- **Step size: -1** – return every single element in the range, going backwards

STRING SLICING

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

EXAMPLE | *Grabbing components of a text string*

```
message = 'I hope it snows tonight!'
message[-1:-9:-1]
```

```
'!thginot'
```

How does this code work?

- **Start: -1** – start with the last element in the string
- **Stop: -9** – stop at the ninth element from the end of the string without including it
- **Step size: -1** – return every single element in the range, going backwards

STRING SLICING

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

EXAMPLE | *Grabbing components of a text string*

```
message = 'I hope it snows tonight!'
message[-9:-1:1]
```

```
' tonight'
```

How does this code work?

- **Start: -9** – start with the ninth element from the end of the string
- **Stop: -1** – stop at last element in the string without including it
- **Step size: 1** – return every single element in the range

STRING SLICING

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

EXAMPLE | *Grabbing components of a text string*

```
message = 'I hope it snows tonight!'
message[-9::]
```

```
' tonight!'
```

How does this code work?

- **Start: -9** – start with the ninth element from the end of the string
- **Stop: 25** – stop at the end of the string by default
- **Step size: 1** – return every single element in the range by default

ASSIGNMENT: STRING SLICING

NEW MESSAGE

February 5, 2022

From: **Alfie Alpine** *(Marketing Manager)*

Subject: **Help Drafting Copy**

Hi there!

I need help drafting some copy.

We have a long customer testimonial and I want to view several options for a good testimonial quote.

The full details are in the file attached.

Looking forward to working with you on this!

marketing_copy.ipynb

Reply Forward

Results Preview

```
print(short_testimonial)
print(happy_customer)
```

```
I love skiing. It's my life.
I love my life!
```

THE LENGTH FUNCTION

The **len()** function returns the number of elements in an iterable

```
# len returns the length of strings
len('snowboard')
```

When used on a string, it returns the number of characters

```
9
```

```
# len will return the length of any iterable object
len(['beginner', 'enthusiast', 'pro'])
```

When used on a list, it returns the number of elements within it (more on lists later!)

```
3
```

ASSIGNMENT: STRING LENGTH

NEW MESSAGE

February 5, 2022

From: **Sierra Schnee** *(Security Consultant)*

Subject: **Cryptography**

Hi, great work last time!

I can't tell you much, but I have attached one more message for you to help me decode using Python.

I think in this batch the length of each message represents the letter in the alphabet for each character in the password.

Can you send me the password once you crack it?

Thanks in advance!

suspicious2.ipynb

Reply Forward

Results Preview

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
print(len('Hello'))
print(alphabet[5])
print(password)
```

```
5
f
maven
```

METHODS

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

Methods are functions that only apply to a specific data type or object

- We call them by following the object with a period then the method name

```
message = 'I hope it snows tonight!'
```

```
message.replace('snow', 'rain')
```

```
'I hope it rains tonight!'
```

*replace is a **string method** being applied to message, which is a string*

```
number = 256
number.replace(5, 9)
```

```
AttributeError: 'int' object has no attribute 'replace'
```

*If you try to apply a string method to an integer, you will receive an **AttributeError***

STRING METHODS

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

find	*Returns the index of the first instance of a given string*	**.find(*string*)**
upper	*Converts all characters in a string to uppercase*	**.upper()**
lower	*Converts all characters in a string to lowercase*	**.lower()**
strip	*Removes leading and trailing spaces, or any other character specified, from a string*	**.strip(*optional string*)**
replace	*Substitutes a given string of characters with another*	**.replace(*string to replace*, *replacement*)**
split	*Splits a list based on a given separator (space by default)*	**.split(*optional string*)**
join	*Joins list elements into a single string separated by a delimiter*	***delimiter*.join(*list*)**

For a list of string methods, visit: ***https://docs.python.org/3/library/stdtypes.html#string-methods***

FIND

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

The **.find()** method searches for a specified value within a string and returns the index of its first instance

```
message = 'I hope it snows tonight!'
```

```
message.find('snow')
```

```
10
```

***'snow'** first appears in the 11th character (index value of 10)*

- **.find(*string*)**

```
message.find('rain')
```

```
-1
```

-1 is returned if the specified value isn't found

UPPER & LOWER

The **.upper()** and **.lower()** methods return a given string with all its characters in uppercase or lowercase respectively

```
message = 'I hope it snows tonight!'
```

```
print(message.upper())
```

```
I HOPE IT SNOWS TONIGHT!
```

```
print(message.lower())
```

```
i hope it snows tonight!
```

STRIP

The **.strip()** method removes leading and trailing spaces *(by default)*, or any other specified character, from a string

- **.lstrip()** removes all leading spaces, and **.rstrip()** removes all trailing spaces

```
message = "      I love the space bar      "

message.strip()
```

```
'I love the space bar'
```

This removes leading and trailing spaces by default

- **.strip(***optional string***)**

```
message = '.Remove the punctuation.'

message.strip('.')
```

```
'Remove the punctuation'
```

This removes leading and trailing periods

REPLACE

The **.replace()** method substitutes a specified string of characters with another

- This is a common method to remove or change punctuation while cleaning text

```
message = 'I hope it snows tonight!'
```

```
message.replace('snow', 'rain')
```

```
'I hope it rains tonight!'
```

*This replaces all instances of **'snow'** with **'rain'***

- **.replace(***string to replace***, *replacement*)**

```
message.replace(' ', '')
```

```
'Ihopeitsnowstonight!'
```

*This replaces all **spaces** with an **empty string**, which creates one continuous word*

SPLIT & JOIN

String Basics

String Arithmetic

Indexing & Slicing

String Methods

F-Strings

The **.split()** method separates a string into a list based on a delimiter *(space by default)*

```
message = 'I hope it snows tonight!'
word_list = message.split(' ')
word_list
```

```
['I', 'hope', 'it', 'snows', 'tonight!']
```

A new, separate element is created each time a space is encountered

- **.split(***optional string***)**

The **.join()** method combines individual list elements into a single string, separated by a given delimiter

```
' '.join(word_list)
```

```
'I hope it snows tonight!'
```

This takes the elements in word_list and combines them into a single string, separated by a single space

- ***delimiter*.join(*list*)**

CHAINING METHODS

Methods can be **chained together** to create powerful expressions

```
message = 'I hope it snows TONIGHT!'
message[6::].lower().split()
```

```
['it', 'snows', 'tonight!']
```

message[6::] → 'it snows TONIGHT!'

'it snows TONIGHT!'.lower() → 'it snows tonight!'

'it snows tonight!'.split() → ['it', 'snows', 'tonight!']

How does this code work?

- The string is **sliced** to remove the first two words ('I hope')
- Then it's converted to all lowercase letters with **.lower()**
- And finally divided into a list of separate words with **.split()**

```
len(message[6::].lower().split())
```

```
3
```

You can even wrap this in a len function to return the number of words after 'I hope'

CHAINING METHODS

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

Methods can be **chained together** to create powerful expressions

```
message = 'I hope it snows tonight!'
```

PRO TIP: Converting text to all upper or lowercase can make many operations, like finding a keyword or joining data much easier

```
message.find('SNOW')
```

```
-1
```

*Find is case-sensitive, so it won't find **'SNOW'** in 'I hope it **snow**s TONIGHT'*

```
message.upper().find('SNOW')
```

```
10
```

By converting the message to uppercase, it finds the match and returns the index

STRING METHOD CHEATSHEET

```
message = 'I hope it snows tonight!'
```

.find(*string*)

Returns the index of the first instance of a given string

```
message.find('snow')
```

```
10
```

.upper()

Converts all characters in a string to uppercase

```
message.upper()
```

```
'I HOPE IT SNOWS TONIGHT!'
```

.lower()

Converts all characters in a string to lowercase

```
message.lower()
```

```
i hope it snows tonight!
```

.strip(*optional string*)

Removes leading or trailing spaces or characters

```
message.strip('!')
```

```
'I hope it snows tonight'
```

.replace(*string to replace*, *replacement*)

Substitutes a string with another

```
message.replace('snow', 'rain')
```

```
'I hope it rains tonight!'
```

.split(*optional string*)

Splits a string into a list based on spaces or a delimiter

```
message.split()
```

```
['I', 'hope', 'it', 'snows', 'tonight!']
```

delimiter.join(*list*)

Combines list elements into a string, separated by a delimiter

```
' '.join(word_list)
```

```
'I hope it snows tonight!'
```

NOTE: There are many more string methods than covered; to review those, or get additional detail on the methods covered, visit the official Python documentation:

https://docs.python.org/3/library/stdtypes.html#string-methods

ASSIGNMENT: STRING METHODS

NEW MESSAGE

February 5, 2022

From: **Alfie Alpine** *(Marketing Manager)*

Subject: **Copy Edits**

We want to draft more copy from the file attached:

- Combine the text messages and call it 'full_text'
- Remove leading spaces and convert to lowercase, call it 'fixed_text'
- Change 'on the mountain' to 'at the ski shop' and call it 'new_text'
- Look up a method to count the number of spaces in 'new_text'; we pay by word so I want to know how much our ad will cost. Word count equals the number of spaces plus 1.

marketing_copy2.ipynb

Reply Forward

Results Preview

```
text1 = '          Your friend Mark'
text2 = 'was'
text3 = 'having'
text4 = 'a great day'
text5 = 'on the mountain'
```

```
full_text
```

```
'          Your friend Mark was having a great day on the mountain'
```

```
fixed_text
```

```
'your friend mark was having a great day on the mountain'
```

```
new_text
```

```
'your friend mark was having a great day at the ski shop'
```

```
word_count
```

```
12
```

F-STRINGS

- String Basics
- String Arithmetic
- Indexing & Slicing
- String Methods
- F-Strings

You can include variables in strings by using **f-strings**

- This allows you to change the output text based on changing variable values

```
name = 'Chris'
role = 'employee'

print(f"{name} is our favorite {role}, of course!")
```

```
Chris is our favorite employee, of course!
```

Add an **f** before the quotation marks to indicate an f-string, and place the variable names into the string by using curly braces **{}**

```
name = 'Anand'
role = 'customer'

print(f"{name} is our favorite {role}, of course!"
```

```
Anand is our favorite customer, of course!
```

Note that the f-string code doesn't change, but the output does change if the variables get assigned new values

ASSIGNMENT: F-STRINGS

NEW MESSAGE

February 5, 2022

From: **Sally Snow** *(Ski Shop Manager)*

Subject: **Price Display Message**

Hi again,

Our website is currently displaying static prices, so we need to manually change them every time a price changes, and write a new one every time we get a new product

Can you write a piece of code for a string that takes product and price as variables? Thanks.

P.S. Don't work with Sierra anymore, she's not an employee

price_display.ipynb

Reply | Forward

Results Preview

```
price = 499.99
product = 'snowboard'

# your print statement here
```

```
The snowboard costs $499.99.
```

```
price = 19.99
product = 'scarf'

# exact copy of print statement in first cell
```

```
The scarf costs $19.99.
```

*These two lines of code should be **identical***

KEY TAKEAWAYS

Strings are used to store **text** and other sequences of characters

- *Analysts interact with strings in categorical data, or in large bodies of text that need to be mined for insight*

Access single characters with **indexing**, and multiple characters with **slicing**

- *Indexing & slicing are used with many other data types beyond strings, so they are critical to master*

Learn and leverage **string methods** to facilitate working with strings

- *You don't need to memorize every method and their syntax on day 1, but it's important to be aware of them*

Use **f-strings** to incorporate variables into your strings

- *This will make your text strings dynamic and allow you to avoid writing repetitive code*