TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Summer 2025 (@ Section 1b | Tuesdays and Thursdays 3PM - 5PM PDT)
Personal Member ID#: **126663**

Unit 2 Cheatsheet

Overview

Here is a helpful cheatsheet outlining common syntax and concepts that will help you in your problem-solving journey! Use this as a reference while you solve the breakout problems for Unit 2. This is not an exhaustive list of all data structures, algorithmic techniques, and syntax you may encounter; it only covers the most critical concepts needed to ace Unit 2. In addition to the material below, you are expected to know any required concepts from previous units.

Standard Concepts

‼️ **This material is in scope for the Standard Unit 2 assessment.**

Built-In Functions

Type Casting

Type casting is the process of transforming or 'casting' data from one type into another type.

`int(x)` Casts a float or string to an integer

- Accepts one parameter `x`: the float or string to turn into an integer
  - Floats will be rounded down
- Returns `x` as an integer.

Example Usage:

```
x = int(2.5) # x will be 2
y = int("5") # y will be 5
```

`float(x)` Cast an integer or string to a floating point number

- Accepts one parameter `x`: the integer or string to turn into an float

- Returns `x` as a float.

Example Usage:

```
x = float(2) # x will be 2.0
y = float("5") # y will be 5.0
z = float("5.3") # z will be 5.3
```

`str(x)` Transform a data type into a string

- Accepts one parameter `x`: a data type, commonly an integer, to turn into a string
    - `x` can also be more complex data types like lists or dictionaries!
- Returns `x` as a string.

Example Usage:

```
x = str(2) # x will be "2"
y = str(2.0) # y will be "2.0"
z = str([1, 2, 3, 4]) # z will be "[1, 2, 3, 4]"
```

Infinity

We can represent positive and negative infinity with the following syntax.

```
positive_infinity = float('inf')

negative_infinity = float('-inf')
```

Infinity is often used as an initial value when finding some unknown value, especially a minimum or maximum value.

Example Usage:

```
lst = [5, 4, 3, 2, 1]

def get_min(lst):
    minimum = float('inf')
    for num in lst:
        if num < minimum:
            minimum = num

    return minimum
```

In the above example, infinity is used to find the minimum value in a list of numbers. We assume the minimum is the largest number possible (infinity), then update our assumption as we iterate through the list and find values lower than infinity.

Infinity is also commonly used as a return value to handle edge cases.

ample Usage:

```python
def safe_divide(a, b):
    if b == 0:
        if a > 0:
            return float('inf')
        else:
            return float('-inf')
    return a / b
```

In the above example infinity is used to gracefully handle cases where a user tries to divide a number by zero. Normally, dividing a number by zero would result in Python raising a `ZeroDivisionError`.

Round Function

`round(number, decimal)` Returns a given number rounded to the specified decimal. **Try it**

- Accepts two parameters:
  - `number`: the number to be rounded. This is a *required* argument.
  - `decimal`: number of decimals to include in the rounded number. This is an *optional* argument. If it is not specified, it defaults to 0 and rounds to the nearest integer.
- Returns the rounded number.

Example Usage:

```python
# Example 1: Round to hundredth
x = 3.14159
rounded = round(x, 2)
print(rounded) # Prints 3.14

# Example 2: Round to nearest whole number
x = 3.14159
rounded = round(x)
print(rounded) # Prints 3
```

Absolute Value Function

`abs(number)` Returns the absolute value of a number. The absolute value is its distance from `0` on a numberline and is mathematically denoted as `|x|`. **Try it**

- Accepts one parameter:
  - `number`: the number to find the absolute value of. This is a *required* argument.
- Returns `number`'s absolute value.

Example Usage:

```python
# Example 1: Absolute value of a positive integer
absolute_value = abs(5)
print(absolute_value) # Prints 5

# Example 2: Absolute value of a negative integer
absolute_value = abs(-5)
print(absolute_value) # Prints 5

# Example 3: Absolute value of a float
absolute_value = abs(-3.14)
print(absolute_value) # 3.14
```

Enumerate

`enumerate(x)` takes an iterable such as a list, dictionary, or string, and adds a counter to the function. It is often used to loop over the indices and values of an iterable simultaneously.

- Accepts two parameters:
    - `x` : an iterable object such as a list, dictionary, or string. This is a *required* parameter.
    - `start` : the value to start the counter at. This is an *optional* parameter. If no default value is specified, the counter will start at 0.
- Returns a sequence of numbers coupled with values in given iterable

Example Usage:

```python

# Example 1: Iterating over indices and characters in a string
my_string = 'code'
for index, char in enumerate(my_string):
  print(index, char)

# Prints:
# 0 c
# 1 o
# 2 d
# 3 e

# Example 2: Enumerate with start value specified
cereals = ['cheerios', 'fruity pebbles', 'cocoa puffs']
for count, cereal in enumerate(cereals, start=1):
  print(count, cereal)

# Prints:
# 1 cheerios
# 2 fruity pebbles
# 3 cocoa puffs
```

Zip

`zip(x, y)` takes two or more iterables and returns a sequence of tuples where each the first item from each iterable forms the first tuple in the sequence, the second item from each iterable forms a second tuple, and so on. It continues until it has placed all elements in the shortest iterable into a function. **Try it**

Zip is useful for iterating over multiple iterables in parallel or for combining data from separate iterables.

- Accepts two or more parameters:
  - `x` : an iterable object such as a list, dictionary, or string.
  - `y` : an iterable object such as a list, dictionary, or string.
  - Optionally accepts additional iterables to zip with `x` and `y`
- Returns a sequence of tuples pairing `x[i]` with `y[i]` where `0 <= i <= min(len(x), len(y))`

Example Usage:

```

# Example 1: Zipping Two Lists
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]
zipped = zip(names, ages)
print(list(zipped)) # Prints [('Alice', 25), ('Bob', 30), ('Charlie', 35)]

# Example 2: Zipping Lists of Different Lengths
names = ['Alice', 'Bob', 'Charlie', 'David']
ages = [25, 30, 35]
zipped = zip(names, ages)
print(list(zipped)) # Prints [('Alice', 25), ('Bob', 30), ('Charlie', 35)]
```

Dictionary Methods & Syntax

Setting and Updating Values

You can add new key-value pairs or update the value of an existing key using the assignment operator `=` . If the key does not exist, a new key-value pair is added to the dictionary.

```
d['d'] = 4          # Adds a new key 'd' with value 4
print(d)            # Outputs: {'a': 1, 'b': 2, 'c': 3, 'd': 4}

d['a'] = 100        # Updates the value of key 'a'
print(d)            # Outputs: {'a': 100, 'b': 2, 'c': 3, 'd': 4}
```

Accessing Elements

Dictionaries in Python allow you to access values using the keys. This can be done primarily in two ways:

ethod 1: Direct access by Key

You can access the value associated with a specific key directly using square brackets `[]`. If the key is not found, this will raise a `KeyError`.

```
d = {'a': 1, 'b': 2, 'c': 3}
print(d['a'])  # Outputs: 1
print(d['b'])  # Outputs: 2
```

Method 2: Using the `get()` Method. Try it

Alternatively, you can use the `get()` method, which provides a safer way to access values. The get() method returns `None` if the key is not found, or a default value that you can specify.

`d.get(key, default_val)` Returns the value of the item in the dictionary `d` with the specified key.

- Accepts two parameters:
  - `key`: the key of the value you want to access. This is a *required* parameter
  - `default_val`: default value to return if the specified key does not exist. This is an *optional* parameter. If no default value is specified, will return `None` for non-existent keys.
- Returns the value of item paired with `key`.

Example Usage:

```
d = {'a': 1, 'b': 2, 'c': 3}
print(d.get('a'))       # Outputs: 1
print(d.get('z'))       # Outputs: None
print(d.get('z', 'Not Found'))  # Outputs: 'Not Found'
```

Pop Method

`d.pop(key, default_val)` Try it

- Accepts two parameters:
  - `key`: the key of the item you want to remove. This is a *required* parameter.
  - `default_val`: default value to return if the specified key does not exist. This is an *optional* parameter. If no default value is specified, will raise an error if no item with specified `key` exists.
- Returns the value of the removed item.

```
d = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

# Example 1: Pop without default_val
d.pop('a') # Returns 1
print(d) #  Prints {'b': 2, 'c': 3, 'd': 4}
d.pop('e') # Raises KeyError


d = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

# Example 2: Get with default_val
d.pop('a', None) # Returns 1
print(d) # Prints {'b': 2, 'c': 3, 'd': 4}
d.pop('e', None) # Returns None
print(d) # Prints {'b': 2, 'c': 3, 'd': 4}
```

Keys Method

`d.keys()` Returns a list of the keys in the dictionary. **Try it**

- Does not have any required parameters
- Returns a list of keys in the specified dictionary.

```
d = {'a': 1, 'b': 2, 'c': 3, 'd': 4}


keys = d.keys()
print(keys) # Prints ['a', 'b', 'c', 'd']
```

Values Method

`d.values()` Returns a list of the values in the dictionary. **Try it**

- Does not have any required parameters
- Returns a list of values in the specified dictionary.

```
d = {'a': 1, 'b': 2, 'c': 3, 'd': 4}


values = d.values()
print(values) # Prints [1, 2, 3, 4]
```

Items Method

`.items()` Returns a list of the key-value pairs in a dictionary. **Try it**

- Does not have any required parameters
- Returns a list of key-value pairs in the specified dictionary. Each key-value pair is represented as a tuple.

```
d = {'a': 1, 'b': 2, 'c': 3, 'd': 4}



items = d.items()
print(items) # Prints [('a', 1), ('b', 2), ('c', 3), ('d', 4)]
```

Python Syntax

Nested Data Structures

Lists and other data structures used to store multiple items in a single variable can be **nested**. This means that lists can store other lists.

The container list is referred to as the **outer list**, while each list nested inside is referred to as an **inner list**.

Example Usage:

```
# Simple List
singers = ["Sabrina Carpenter", "FKA Twigs", "Elliot Smith"]

# Nested List
albums = [
            ["Sabrina Carpenter", "Short n' Sweet"],
            ["FKA Twigs", "Magdalene"],
            ["Elliot Smith", "Either/Or"]]

# Nested Lists Where Inner Lists Have Unequal Length
numbers = [
            [1],
            [1, 2],
            [1, 2, 3]]

# Triply Nested List
water_levels = ["Shallow", ["Deep", ["Deeper"]]]
```

You may also hear lists referred to by their dimensions. A simple list like `singers` in the above example may be referred to as a **1D** or 1-Dimensional list. A list of lists like `albums` or `numbers` may be referred to as a **2D list**, and so on.

Nested lists where each inner list has the same length are often referred to as **matrices**.

Example Usage:

```
matrix = [
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]]
```

We can access and modify each inner list using normal list indexing syntax.

Example Usage:

```
# Example 1: Retrieving Album Data
albums = [
            ["Sabrina Carpenter", "Short n' Sweet"],
            ["FKA Twigs", "Magdalene"],
            ["Elliot Smith", "Either/Or"]]

first_album = albums[0]
print(first_album) # Output: ["Sabrina Carpenter", "Short n' Sweet"]

# Example 2: Updating a Row in a Matrix
matrix = [
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]]

matrix[2] = [100, 200, 300]
print(matrix) # Output: [[1, 2, 3], [4, 5, 6], [100, 200, 300]]
```

We can use **multiple indices** to access and modify elements within the inner lists.

Example Usage:

```


# Example 1: Retrieving a Singer
albums = [
            ["Sabrina Carpenter", "Short n' Sweet"],
            ["FKA Twigs", "Magdalene"],
            ["Elliot Smith", "Either/Or"]]

fka_twigs = albums[1][0]
print(fka_twigs) # Output: FKA Twigs

# Example 2: Updating a cell in a matrix
matrix = [
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]]
matrix[1][1] = "Surprise!"
print(matrix) # Output: [[1, 2, 3], [4, 'Suprise!', 6], [7, 8, 9]]
```

We can also have nested dictionaries or nest dictionaries inside of lists and vice versa.

Example Usage:

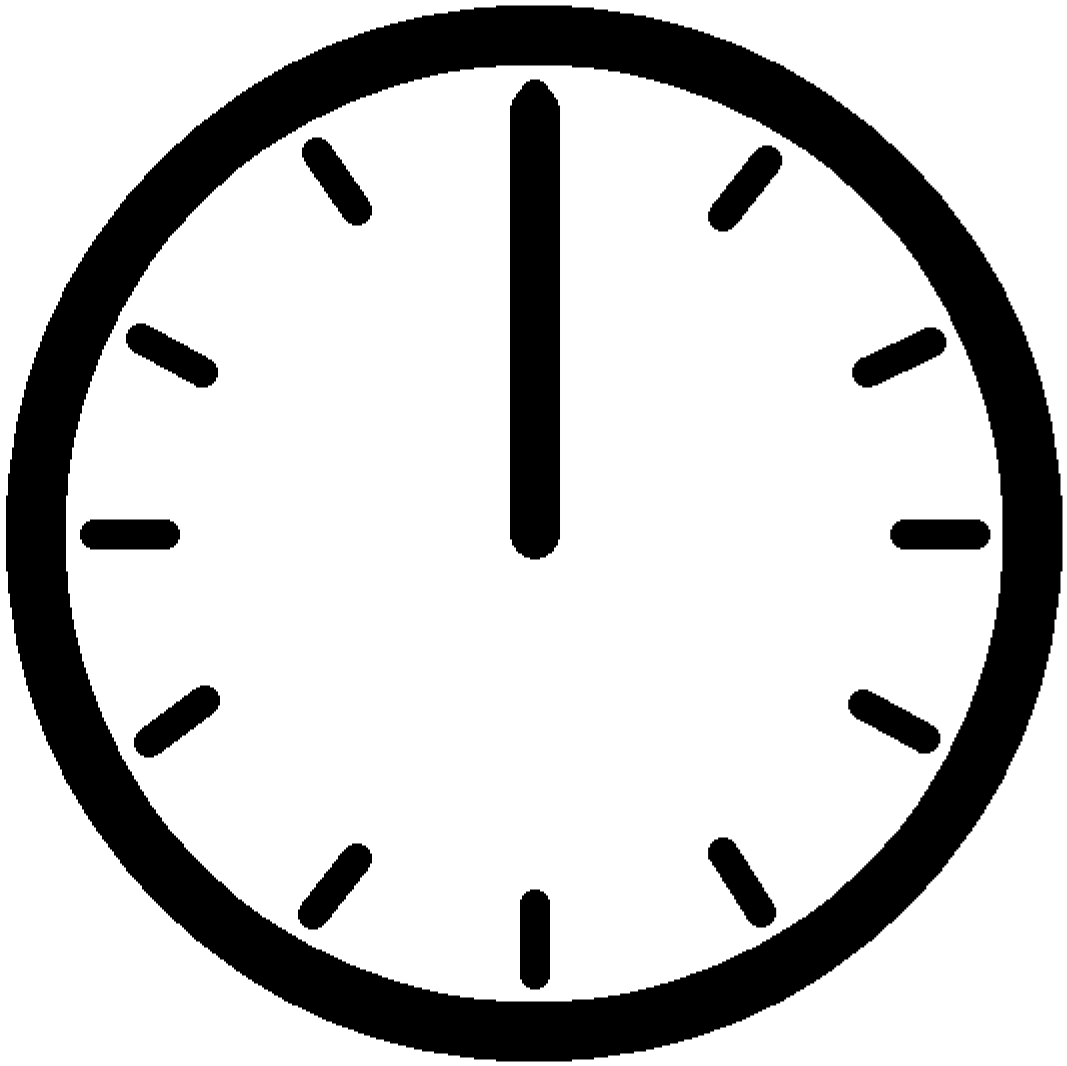
```python
# Example 1: Nested Dictionaries
address_book = {
    "John Doe": {
        "phone": "555-1234",
        "email": "johndoe@example.com",
        "address": {
            "street": "123 Maple Street",
            "city": "Springfield",
            "state": "IL",
            "zip": "62701"
        }
    },
    "Jane Smith": {
        "phone": "555-5678",
        "email": "janesmith@example.com",
        "address": {
            "street": "456 Oak Avenue",
            "city": "Shelbyville",
            "state": "IL",
            "zip": "62565"
        }
    }
}

address_book["John Doe"]["email"]
john_email = address_book["John Doe"]["email"]
print(john_email) # Output: 'johndoe@example.com'

# Example 2: List of Dictionaries
students = [
    {
        "name": "John Doe",
        "age": 16,
        "grade": "11th",
        "favorite_subject": "Math"
    },
    {
        "name": "Jane Smith",
        "age": 17,
        "grade": "12th",
        "favorite_subject": "English"
    },
    {
        "name": "Emily Johnson",
        "age": 16,
        "grade": "11th",
        "favorite_subject": "Biology"
    }
]

jane_age = students[1]["age"]
print(jane_age) # Output: 17
```

Nested Loops

Just as a nested list means a list within a list, a **nested loop** means a loop within a loop! They allow us to perform repeated actions within repeated actions.

When we nest loops, the inner loop will run completely every time the outer loop runs once.

Imagine the outer loop as the short hour hand of an analog clock, and the inner loop as the longer minute hand. The minute hand has to complete a full rotation around the clock before the hour hand increments once.



Example Usage:

```python
for i in range(1, 4):
    print("Outer loop incremented")
    for j in range(1, 4):
        print(f"i = {i}, j = {j}")

# Output:
# i = 1, j = 1
# i = 1, j = 2
# i = 1, j = 3
# i = 2, j = 1
# i = 2, j = 2
# i = 2, j = 3
# i = 3, j = 1
# i = 3, j = 2
# i = 3, j = 3
```

Nested loops are often used to iterate over nested lists and matrices.

Example Usage:

```python
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Using nested loops to iterate over the matrix
for row in matrix:
    for item in row:
        print(item, " ")
    print()  # Print a new line after each row

# Output:
# 1 2 3
# 4 5 6
# 7 8 9
```

A nested for loop is the most common example of a nested loop, but we can also nest other types of loops. We can nest while loops together or nest while loops within for loops and vice versa.

Example Usage:

```
numbers = [3, 5, 2]

# Outer for loop iterates over each number in the list
for num in numbers:
    print(f"Counting down from {num}:")

    # Inner while loop counts down from the current number to 0
    while num >= 0:
        print(num)
        num -= 1  # Decrement the number by 1 each time

    print("---")  # Separator to indicate moving to the next number

# Output:
# Counting down from 3:
# 3
# 2
# 1
# 0
# ---
# Counting down from 5:
# 5
# 4
# 3
# 2
# 1
# 0
# ---
# Counting down from 2:
# 2
# 1
# 0
# ---
```

⚠️ While nested loops are powerful, they do have some drawbacks:

- **Performance:** Nested loops can significantly increase the time it takes your code to run, especially when used on large inputs or when deeply nested.
- **Readability** Deeply nested loops, like triply or quadrupaly nested lists can make code difficult to read. We should always consider whether there is an alternative solution before nesting loops deeply.

Sets

Sets are a built-in data structure in Python that represent an unordered collection of unique elements. They are often used to track seen values, eliminate duplicates, and find overlap between multiple pieces data.

Sets maintain the following characteristics:

- **Unordered**: Sets do not maintain any particular order of elements (i.e. they are not indexed like lists or strings).
- **Unique elements**: Every element in a set must be unique. If a duplicate value is added to a set, the set will automatically remove the duplicate.
- **Mutable**: Sets can be modified. Values can be added or removed without needing to make a new set.
- **Iterable**: Sets can be iterated over using a loop

A new set can be created using curly braces `{}` or using the `set()` function.

Example Usage:

```
# Create a new Set
my_set = {1, 2, 3, 4}

# Using the set() function
another_set = set([1, 2, 3, 4])

# Creating an empty set
empty_set = set()  # Note: {} creates an empty dictionary, not a set
```

We can operate on a set using the following basic functions:

- **`add()`**: Adds an element to the set.
- **`remove()`**: Removes an element from the set. Raises a `KeyError` if the element is not found.
- **`discard()`**: Removes an element if it is present, without raising an error if it is not found.
- **`clear()`**: Removes all elements from the set.

Example Usage:

```
my_set = {1, 2, 3}

my_set.add(4)       # {1, 2, 3, 4}
my_set.remove(2)    # {1, 3, 4}
my_set.remove(5)    # Raises KeyError
my_set.discard(5)   # {1, 3, 4} - No error if element not found
my_set.clear()      # {}
```

Sets support various mathematical operations such as union, intersection, difference, and symmetric difference

- **Union: `a | b`**: Returns the set of elements contained in *either* set `a` or set `b`.
- **Intersection: `a & b`**: Returns the set of elements contained in *both* set `a` or set `b`.
- **Difference: `a - b`**: Returns the set of elements contained in set `a` *but not in* set `b`.

- **Symmetric Difference: `a ^ b`**: Returns the set of elements contained in *either* set `a` or set `b` but *not in both*.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Union: Elements in either set
union_set = set1 | set2          # {1, 2, 3, 4, 5}

# Intersection: Elements common to both sets
intersection_set = set1 & set2    # {3}

# Difference: Elements in set1 but not in set2
difference_set = set1 - set2      # {1, 2}

# Symmetric Difference: Elements in either set, but not both
symmetric_difference_set = set1 ^ set2  # {1, 2, 4, 5}
```

Advanced Concepts

‼️ **This material is in scope for the Advanced Unit 2 assessment.** The standard material above is also in scope for advanced assessments.

Built-In Functions

Sorted

`sorted(x)` Returns a sorted list of the iterable `x` **Try it**

- Accepts three parameters:
  - `x`: an iterable such as a list, dictionary, tuple, etc. to sort. This is a *required* parameter.
  - `key`: a function that specifies a custom sorting order. This is an *optional* parameter. If no key is specified, items in the iterable are sorted in ascending order by default.
  - `reverse`: `True` or `False`. If `True`, items will be sorted in descending order. This is an *optional* parameter. By default, `reverse` has a value of `False`.
- Returns `x` as a sorted list

Example Usage:

```python
# Example 1: Sorting a List in Ascending Order
lst = [1, 5, 3]
result = sorted(lst)
print(result) # Output: [1, 3, 5]

# Example 2: Sorting a List in Descending Order
lst = [1, 5, 3]
result = sorted(lst, reverse = True)
print(result) # Output: [5, 3, 1]

# Example 3: Sorting Keys in a Dictionary
my_dict = {'apple': 2, 'banana': 3, 'cherry': 1}
result = sorted(my_dict)
print(result)  # Output: ['apple', 'banana', 'cherry']

# Example 4: Sorting Strings by Length
words = ["apple", "orange", "banana", "grape"]
result = sorted(words, key=len)
print(result)  # Output: ['apple', 'grape', 'orange', 'banana']

# Example 5: Sorting By Last Character With a Custom Function
def last_character(s):
    return s[-1]

words = ["apple", "banana", "cherry", "date"]
result = sorted(words, key=last_character)
print(result)  # Output: ['banana', 'apple', 'date', 'cherry']
```

Python Syntax

Lambda Functions

`lambda arg1, arg2, etc. : expression` anonymous function that returns the result of evaluating expression on `arg1` , `arg2` , etc. Try It

A lambda function is an *anonymous* function, meaning it is a function without a name.

Example Usage:

```python
# Example 1: Lambda Function with 1 Argument
return_value = lambda x : x + 10
print(return_value(100)) # Prints 110

# Example 2: Lambda Function with Multiple Arguments
return_value = lambda a, b: a + b
print(return_value(10, 20)) # Prints 30
```

Lambda functions are often used with the `sorted()` function to specify a custom sort key. Previously, we've seen that we can create a function to sort an iterable in a custom way.

xample Usage:

```python
# Example 1: Sorting By Last Character With a Custom Function (No Lambda)
def last_character(s):
    return s[-1]

words = ["apple", "banana", "cherry", "date"]
result = sorted(words, key=last_character)
print(result)  # Output: ['banana', 'apple', 'date', 'cherry']
```

With a lambda function, we can shorten this to:

```python
words = ["apple", "banana", "cherry", "date"]
result = sorted(words, key=lambda x: x[-1])
print(result)
```

Ternary Operators

A ternary operator is special shorthand syntax that allows us to write simple if-else conditions on a single line.

The general syntax for a single line if-else statement is:

```python
value_if_true if condition else value_if_false
```

Example Usage:

```python
a = 10
b = 20

# ternary operator
max_value = a if a > b else b

# normal conditional syntax
if a > b:
    max_value = a
else:
    max_value = b
```

Dictionary Comprehensions

`result_list = {key_expression: value_expression for item in iterable}` makes a dictionary pairing `key_expression` with `value_expression` on each element in an iterable and stores the result of each evaluation in `result_list`.

Just as we can create lists based on values in other lists using a list comprehension, we can create new dictionaries based off of values in other dictionaries, lists or strings.

Example Usage:

```
# Example 1: Map Integers to Their Square
lst = [1, 2, 3, 4, 5, 6]
squares = {x: x**2 for x in lst}
print(squares) # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}

# Example 2: Converting List of Tuples to a Dictionary
pairs = [('a', 1), ('b', 2), ('c', 3)]
dictionary = {key: value for key, value in pairs}
print(dictionary)

# Exmaple 3: Even Squares:
even_squares = {x: x**2 for x in range(1, 11) if x % 2 == 0}
print(even_squares) # {2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

Tuples

A **tuple** is a data type used to store multiple items in a single variable. In most cases tuples are used to group together 2-3 items. Tuples that store 3 items are sometimes referred to as a 'triple'. Tuples are commonly used to store pairs of data together or return multiple values inside a function. Try it

Tuples are defined using round brackets.

Example usage:

```
my_tuple = ("Mario", "Luigi")
print(my_tuple) # Prints ("Mario", "Luigi")
```

Like lists and strings, we can use indices to access elements of a tuple

Example Usage:

```
my_tuple = ("Mario", "Luigi")
mario = my_tuple[0]
luigi = my_tuple[1]
print(mario) # Prints "Mario"
print(luigi) # Prints "Luigi"
```

Like strings, tuples are immutable, meaning we cannot update the contents of a tuple. If we update the contents of a tuple, we create a new tuple.

Example usage:

```
my_tuple = (10, 20)
my_tuple[0] = 30 # Results in TypeError: 'tuple' object does not support item assignment
```

Bonus Syntax & Concepts

The following concepts are nice to know and may improve your code readability or help you solve certain problems more easily and efficiently. However, they are not *required* to solve any of the problems in this unit. These concepts are **not in scope for either the Standard or Advanced Unit 2 assessments**, and you do not need to memorize them! Click on each concept to read more about how to use it.

- **`remove(x)`** Removes first element with value `x` from list.
- **`defaultdict()`** Creates a dictionary with default values for new or missing keys. Helps to avoid `KeyError`s!
- **`Counter(x)`** Creates a dictionary that stores elements of a given iterable `x` as keys and their counts as values. Essentially creates a frequency dictionary for you!
- **`d.copy()`** Returns a copy of the dictionary `d`.
- What's the difference between `sorted()` and `sort()`?