Starter Activity.

Code it:

Write a program that will ask for a number of days and then will show how many hours, minutes and seconds are in that number of days.

Starter Activity.

Answer

```
days = int(input("Enter the number of days: "))
hours = days*24
minutes = hours*60
seconds = minutes*60
print("In", days, "days there are...")
print(hours, "hours")
print(minutes, "minutes")
print(seconds, "seconds")
```

```
In 3 days there are...
72 hours
4320 minutes
259200 seconds
```

techtalent academy

TechTalent Academy

Data Science Course

Python Fundamentals Part 2

Lesson Objectives.

- More Operators
- IF Statements
- Lists and Tuples
- Sets and Dictionaries

Comparison Operators.

Operator	Description	Python example
<	Is less than	**if** age < 12:
<=	Is less than or equal to	**if** age <= 12:
>	Is greater than	**if** age > 12:
>=	Is greater than or equal to	**if** age >= 12:
==	Is equal to	**if** age == 12:
!=	Is not equal to	**if** age != 12:

Comparison Operators.

Here the comparison operators are being used to compare two values x and y

One condition is tested

```
x = 5
y = 3

print(x == y)
print(x != y)
print(x > y)
print(x < y)
print(x >= y)
print(x <= y)
```

output

```
False
True
True
False
True
False
```

Logical Operators.

To test more than one condition

Operator	Description
and	Returns True if both conditions are met
or	Returns True if either or both conditions are met
not	A true expression becomes false and vice versa

Comparison and logical operators combined.

Two conditions are tested

Here the logical operators are being used to combine conditional statements

x=5
y=3

```
#Returns True if both statements are true
x <5 and x <10
```

✓ 0.7s

```
False
```

```
#Returns True if one of the statements is true
x <5 or x <4
```

✓ 0.3s

```
False
```

```
#Reverses the result and will return False is the result is not true
not(x <5 and X < 10)
```

✓ 0.5s

```
True
```

Selection.

Selection is used to choose between two or more options in programming we use an **IF STATEMENT**

Conditional → IF you sleep early

Results → THEN you will wake up early

Results → ELSE you will wake up late

IF conditional statement.

Structure:

```
if condition to test is met:
        execute code 1
else:
        execute code 2
```

indentation needed

```
x=5
if x==5:
    print("this is correct")
else:
    print("this is incorrect")
```

✓ 0.4s

```
this is correct
```

```
x=5
if x==7:
    print("this is correct")
else:
    print("this is incorrect")
```

✓ 0.4s

```
this is incorrect
```

IF and ELIF.

In addition to the **IF** statement, you can use **ELIF to continue to build conditional statements** into your code underneath your original IF statement.

IF Condition 1 met:
Perform operation 1

ELIF Condition 2 met:
Perform operation 2

ELIF Condition 3 met:
Perform operation 3

ELSE:
Perform operation 4

```
mark = int(input("Enter mark: "))

if mark > 75:
    print("Merit")
elif mark > 65:
    print("Pass")
else:
    print("Fail")
```

✓ 2.4s

```
Fail
```

Task: If statements .

Ask the user to enter a number between 10 and 20. If they enter a number within this range, display the message “Thank you”, otherwise display the message “Incorrect answer”.

Nested IF.

It is possible to nest IF statements to handle more complex logic

```
examlevel = int(input("Enter exam level: "))

if examlevel == 3:
    mark = int(input("Enter level 3 mark: "))
    if mark > 65:
        print("Pass")
    else:
        print("Fail")

elif examlevel == 4:
    mark = int(input("Enter level 4 mark: "))
    if mark > 50:
        print("Pass")
    else:
        print("Fail")
else:
    print("Invalid Level")
```

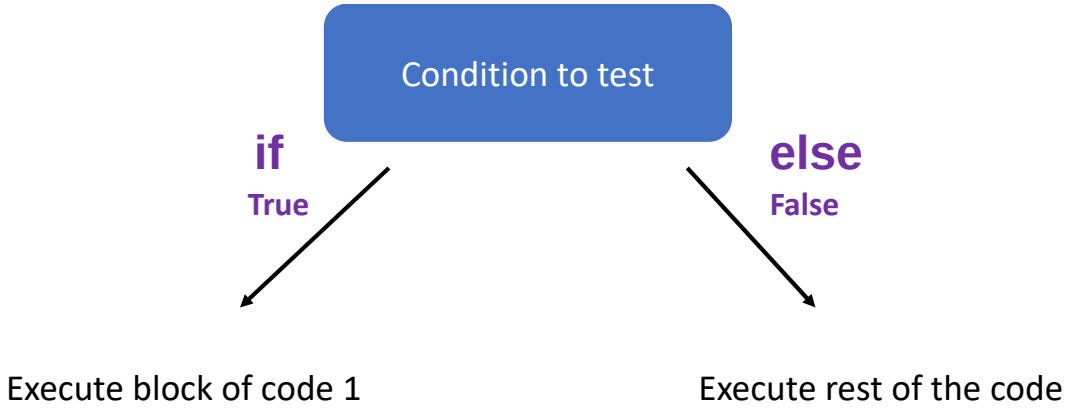
IF with multiple conditions.

Combine conditions to handle more complex logic using **and, or, not**

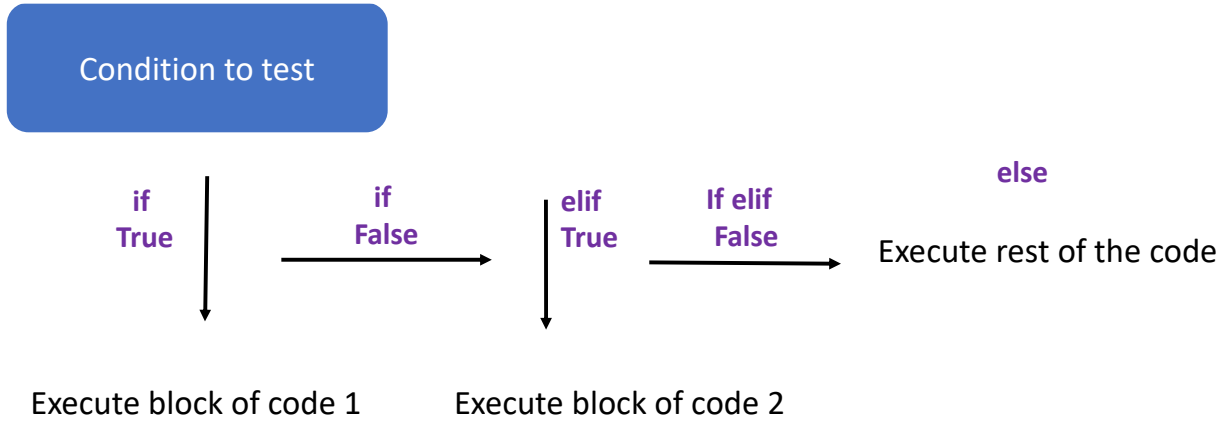
```
if examlevel == 1 or examlevel == 2:
    mark = int(input("Enter Level 1 or 2 mark: "))
    if mark > 75:
        print("Pass")
    else:
        print("Fail")
elif examlevel == 3 or examlevel == 4:
    mark = int(input("Enter Level 3 or 4 mark: "))
    if examlevel == 3 and mark > 65:
        print("Pass")
    elif examlevel == 4 and mark > 50:
        print("Pass")
    else:
        print("Fail")
else:
    print("Invalid Level")
```

IF with multiple conditions.

Situation 1: test one if/else statement



Situation 2: test more than one conditions with elif



Task: Nested If statements.

Ask the user if it is raining and convert their answer to lower case so it doesn't matter what case they type it in. If they answer "yes", ask if it is windy. If they answer "yes" to this second question, display the answer "It is too windy for an umbrella", otherwise display the message "Take an umbrella". If they did not answer yes to the first question, display the answer "Enjoy your day".

[ ]

Lists.

- **Having a variable which can only contain a single value can be quite limiting.**
- **It can be useful for a variable to refer to a collection of data. Lists refer to multiple values, which are all contained and accessible through a single variable**

```
farm_cows = ["Winnie the Moo", "Dasiy", "Milkshake", "Buttercup"]

print(farm_cows[2])
✓ 0.4s
Milkshake
```

- **To access the Milkshake string, index 2 is referenced with a pair of bracket [ ]**
- **The first value ("Winnie the Moo") would be accessed by farm_cows[0], this is because indexes start counting 0.**

[ ]

Lists.

negative index:	-4	-3	-2	-1
index:	0	1	2	3

class_list= ["Caroline", "Lea", ”Mike", "Ismael"]

List= [item1, item2, item3]

“string” or number

class_list[0]

Caroline

[ ]

Lists Methods.

Lists have multiple methods to manipulate the current state of the list:

```
farm_cows = ["Winnie the Moo", "Dasiy", "Milkshake", "Buttercup"]
farm_cows.pop(1)
print(farm_cows)
```

✓ 0.5s

```
['Winnie the Moo', 'Milkshake', 'Buttercup']
```

- **.pop(index)** – **remove an item at the specified index**
- **.insert(index, value)** – **add the value to the list at the specified index**
- **.append(value)** – **add value to the end of the list**
- **.remove(value)** – **remove the first instance of a specified value from a list**
- **.sort()** – **sort the list**
- **len(list_variable)** – **will provide the length of the list**

[ ]

Lists Subset.

A subsection of a list can be taken using the square brackets:

```
farm_cows = ["Winnie the Moo", "Dasiy", "Milkshake", "Buttercup"]
print(farm_cows[2])
print(farm_cows[1:3])
print(farm_cows[1:])
print(farm_cows[:3])
```

✓ 0.3s

```
Milkshake
['Dasiy', 'Milkshake']
['Dasiy', 'Milkshake', 'Buttercup']
['Winnie the Moo', 'Dasiy', 'Milkshake']
```

Entering a single value will return the value at that index in the list.

Using the colon: (,) a range of value can be taken to create a new list.

[ ]

Multidimensional Lists.

A single list is useful for interacting with a row of data.

A multi dimensional list (a list made up of lists) is useful for interacting with a table of data

```
farm_cows = ["Winnie the Moo", "Dasiy", "Milkshake", "Buttercup"]
farm_sheep = ["Baart", "Barbara"]
farm_pigs = ["Bacon", "Hamlet", "Hog"]

farm_animals = [farm_cows, farm_sheep, farm_pigs]

print(farm_animals [2] [1])
print(farm_animals [0] [3])
```

✓ 0.7s

```
Hamlet
Buttercup
```

[ ]

Loosely Typed Lists.

A list of items can contain any data types, it can be a list that contains lists, strings, numbers etc.

```
farm = ["Snow Hiil Farm", ["Winnie the Moo", "Daisy"], 1991]
#         string              List                     Int

print(farm[0])
print(farm[1])
print(farm[2])
```

✓ 0.8s

```
Snow Hiil Farm
['Winnie the Moo', 'Daisy']
1991
```

[ ]

List methods.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Task: Lists.

Create a list of two sports. Ask the user what their favourite sport is and add this to the end of the list. Sort the list and display it.

( )

Tuples.

A tuple is a read-only sequence of items separated by commas and enclosed within parentheses.

Tuples cannot be changed once they have been created.

```
cat_list = ["Buttons", 23, 4.0]
cat_tuple = ("Mr Socks", 23, 4.0)
```

0.3s

```
# name cannot be changed
cat_tuple[0] = "Twiglet"
print(cat_tuple)
```

0.6s

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-471671a444a9> in <module>
----> 1 cat_tuple[0] = "Twiglet"
      2 print(cat_tuple)

TypeError: 'tuple' object does not support item assignment
```

( )

Tuples.

index: 0 1 2

country_list= ("UK", "Bolivia", "Australia")

⬇

country_list[0]

⬇

UK

( )

Tuple Methods.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

{ }

Sets.

A set is used to store multiple items in a single variable. A set is a collection which is unordered, unindexed, items can be added or removed, items are unique.

```
cat_set = {"Buttons", "Mr Socks", "Twiglet"}
print(cat_set)
```

✓ 3.5s

```
{'Twiglet', 'Buttons', 'Mr Socks'}
```

{ }

techtalent academy

Set Methods.

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

{ }

techtalent academy

Dictionaries.

Dictionaries are similar to lists such that they can store a collection of values, but you usually don't use indexes to find the items.

Each value is stored with a key. When the key is searched for in the dictionary with a bracket [ ]. , the value is returned.

```
farm_animals = {"cow": "Cows go moo",
                "sheep": "Sheep go baa",
                "cat": "Cat go meow"
                }

print(farm_animals["cow"])
print(farm_animals["sheep"])
print(farm_animals["cat"])
```

✓ 0.9s

```
Cows go moo
Sheep go baa
Cat go meow
```

{ }

Dictionaries.

“string” or number

dictionary= {key: value}

dic_TTA= {"email": "https://techtalent.academy", "city": "Birmingham”}

{ }

techtalent academy

Adding Items To An Empty Dictionary.

Keys and values do not need to be initialised with the dictionary.

```
#adding to dictionary
farm_animals = {}
farm_animals["cow"] = "Cows go moo"
farm_animals["sheep"] = "Sheep goes baa"

print(farm_animals["cow"])
print(farm_animals["sheep"])
```

✓ 0.6s

```
Cows go moo
Sheep goes baa
```

{ }

techtalent academy

Keys and values.

A list of keys and/or values can be retrieved using the .keys() and .values() methods within a dictionary

```
# keys and values
farm_animals = {"cow" : "Cows go moo",
                "sheep": "Sheep goes baa",
                "cat": "Cats go meow"
               }

list(farm_animals.keys())
list(farm_animals.values())
```

✓ 0.1s

```
['Cows go moo', 'Sheep goes baa', 'Cats go meow']
```

{ }

techtalent academy

Nested Dictionaries.

A dictionary can contain dictionaries, this is called nested dictionaries.

```python
mypets = {
  "pet1" : {
    "name" : "Fig",
    "year" : 2004
  },
  "pet2" : {
    "name" : "Ruby",
    "year" : 2007
  },
  "pet3" : {
    "name" : "Twiglet",
    "year" : 2011
  }
}

print(mypets)
```

✓ 0.4s Python

```
{'pet1': {'name': 'Fig', 'year': 2004}, 'pet2': {'name': 'Ruby', 'year': 2007}, 'pet3':
{'name': 'Twiglet', 'year': 2011}}
```

{ }

Dictionary Methods.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Data Types Recap.

List = [item1, “item2”, item3]

Dictionary = { key1 : value1, key2 : value2, “key3” : value3}

Tuples = (item1, item2, “item3”)

Sets = {item1, item2, “item3”}

An item can be an integer (whole number), a float (numbers with decimals), or a “string”

Plenary.

Change the value from "apple" to "kiwi", in the `fruits` list.

```
fruits = ["apple", "banana", "cherry"]
          =
```

Use the correct syntax to print the number of items in the `fruits` tuple.

```
fruits = ("apple", "banana", "cherry")
print(          )
```

Use the `get` method to print the value of the "model" key of the `car` dictionary.

```
car =   {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(                )
```