All even numbers between 1 and 20.

```
for num in range(2, 21, 2):
    print(num)
```

• Change List Items

To change the value of a specific item, refer to the index number:

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```

Change a Range of Item Values

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi",
"mango"]

thislist[1:3] = ["blackcurrant", "watermelon"]

print(thislist)
```

If you insert *more* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

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

thislist[1:2] = ["blackcurrant", "watermelon"]

print(thislist)
```

Change the second and third value by replacing it with *one* value:

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

thislist[1:3] = ["watermelon"]

print(thislist)
```

Insert Items

The insert () method inserts a new list item without replacing any of the existing values.

The `insert()` method inserts an item at the specified index:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```

- # Add List Items

Append Items

To add an item to the end of the list, use the `append()` method:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

Insert Items

To insert a list item at a specified index, use the `insert()` method.

The `insert()` method inserts an item at the specified index:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Extend List

To append elements from *another list* to the current list, use the `extend()` method.

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

Add Any Iterable

The `extend()` method does not have to append *lists*, you can add any iterable object (tuples, sets, dictionaries etc.).

```
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
thislist.extend(thistuple)
print(thislist)
```

Remove List Items

Remove Specified Item

The `remove()` method removes the specified item.

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

Remove Specified Index

The `pop()` method removes the specified index.

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

The `del` keyword also removes the specified index:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

Clear the List

The `clear()` method empties the list.

The list remains, but it has no content.

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

thislist.clear()

print(thislist)
```

• Python If ... Else

Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the `if` keyword.

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

In this example we use two variables, `a` and `b`, as part of the if statement to test whether `b` is greater than `a`. As `a` is `33`, and `b` is `200`, we know that 200 is greater than 33, so we print to the screen that "b is greater than a".

Indentation

Python relies on indentation (whitespace at the beginning of a line) to define the scope of the code. Other programming languages often use curly brackets for this purpose.

```
a = 33

b = 200

if b > a:

print("b is greater than a") # you will get an error
```

Elif

The `elif` keyword is Python's way of saying "If the previous conditions were not true, then try this condition".

```
a = 33

b = 33

if b > a:

  print("b is greater than a")

elif a == b:

  print("a and b are equal")
```

In this example `a` is equal to `b`, so the first condition is not true, but the `elif` condition is true, so we print to screen that "a and b are equal".

Else

The `else` keyword catches anything which isn't caught by the preceding conditions.

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

In this example `a` is greater than `b`, so the first condition is not true, also the `elif` condition is not true, so we go to the `else` condition and print to screen that "a is greater than b".

You can also have an `else` without the `elif`:

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

One line if statement:

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

One line if else statement:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

You can also have multiple else statements on the same line:

One line if else statement, with 3 conditions:

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

And

The `and` keyword is a logical operator, and is used to combine conditional statements:

Test if `a` is greater than `b`, AND if `c` is greater than `a`:

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

Or

The `or` keyword is a logical operator, and is used to combine conditional statements:

Test if `a` is greater than `b`, OR if `a` is greater than `c`:

```
a = 200

b = 33

c = 500

if a > b or a > c:

  print("At least one of the conditions is True")
```

Not

The `not` keyword is a logical operator, and is used to reverse the result of the conditional statement:

Test if `a` is NOT greater than `b`:

```
a = 33

b = 200

if not a > b:

  print("a is NOT greater than b")
```

Nested If

You can have `if` statements inside `if` statements, this is called *nested* `if` statements.

```
x = 41

if x > 10:

  print("Above ten,")

  if x > 20:

    print("and also above 20!")

  else:

    print("but not above 20.")
```

The pass Statement

`if` statements cannot be empty, but if you for some reason have an `if` statement with no content, put in the `pass` statement to avoid getting an error.

```
a = 33

b = 200

if b > a:

  Pass
```

Loop Lists

Loop Through a List

You can loop through the list items by using a `for` loop:

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

for x in thislist:

  print(x)
```

Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.

Use the `range()` and `len()` functions to create a suitable iterable.

Print all items by referring to their index number:

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

for i in range(len(thislist)):

  print(thislist[i])
```

Tuples

Tuple

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

A tuple is a collection which is ordered and unchangeable.

Tuples are written with round brackets.

```
thistuple = ("apple", "banana", "cherry")

print(thistuple)
```

Tuple Items

Tuple items are ordered, unchangeable, and allow duplicate values.

Tuple items are indexed, the first item has index `[0]`, the second item has index `[1]` etc.

Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates

Since tuples are indexed, they can have items with the same value:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")

print(thistuple)
```

Tuple Length

To determine how many items a tuple has, use the `len()` function:

```
thistuple = ("apple", "banana", "cherry")

print(len(thistuple))
```

Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

One item tuple, remember the comma:

```
thistuple = ("apple",)

print(type(thistuple))


#NOT a tuple

thistuple = ("apple")

print(type(thistuple))
```

Tuple Items - Data Types

Tuple items can be of any data type:

String, int and boolean data types:

```
tuple1 = ("apple", "banana", "cherry")

tuple2 = (1, 5, 7, 9, 3)

tuple3 = (True, False, False)
```

A tuple can contain different data types:

A tuple with strings, integers and boolean values:

```
tuple1 = ("abc", 34, True, 40, "male")
```

type()

From Python's perspective, tuples are defined as objects with the data type 'tuple':

```
mytuple = ("apple", "banana", "cherry")

print(type(mytuple))
```

The tuple() Constructor

It is also possible to use the `tuple()` constructor to make a tuple.

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets

print(thistuple)
```

Python Collections (Arrays)

There are four collection data types in the Python programming language:

- **List is a collection which is ordered and changeable. Allows duplicate members.**
- **Tuple is a collection which is ordered and unchangeable. Allows duplicate members.**
- **Set is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.**
- **Dictionary is a collection which is ordered** and changeable. No duplicate members.**

Access Tuple Items

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

```
thistuple = ("apple", "banana", "cherry")

print(thistuple[1])
```

Negative Indexing

Negative indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item etc.

```
histuple = ("apple", "banana", "cherry")

print(thistuple[-1])
```

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

This example returns the items from the beginning to, but NOT included, "kiwi":

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi",
"melon", "mango")

print(thistuple[:4])
```

By leaving out the end value, the range will go on to the end of the tuple:

This example returns the items from "cherry" and to the end:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi",
"melon", "mango")

print(thistuple[2:])
```

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

This example returns the items from index -4 (included) to index -1 (excluded)

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi",
"melon", "mango")

print(thistuple[-4:-1])
```

Check if Item Exists

To determine if a specified item is present in a tuple use the `in` keyword:

Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")

if "apple" in thistuple:

  print("Yes, 'apple' is in the fruits tuple")
```

Update Tuples

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.

But there are some workarounds.

Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

```
x = ("apple", "banana", "cherry")

y = list(x)

y[1] = "kiwi"

x = tuple(y)



print(x)
```

Add Items

Since tuples are immutable, they do not have a built-in `append()` method, but there are other ways to add items to a tuple.

1. Convert into a list: Just like the workaround for *changing* a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

Convert the tuple into a list, add "orange", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")

y = list(thistuple)

y.append("orange")

thistuple = tuple(y)
```

2. Add tuple to a tuple. You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

Create a new tuple with the value "orange", and add that tuple:

```
thistuple = ("apple", "banana", "cherry")

y = ("orange",)

thistuple += y

print(thistuple)
```

Remove Items

Note: You cannot remove items in a tuple.

Tuples are unchangeable, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

Convert the tuple into a list, remove "apple", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")

y = list(thistuple)

y.remove("apple")

thistuple = tuple(y)
```

Or you can delete the tuple completely:

The `del` keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")

del thistuple

print(thistuple) #this will raise an error because the tuple
no longer exists
```


Loop Lists
List Comprehension
Sort Lists
Copy Lists
Join Lists
List Methods

Unpack Tuples
Loop Tuples
Join Tuples
Tuple Methods