# Turtle Graphics, Casting, Comparison Operators and Loops

APCO/IASC 1P00 – Lecture 02 (Winter 2023)

Brock University

Maysara Al Jumaily
amaysara@brocku.ca

January 17, 2023

# Lecture Outline

# Basic Data Types

- `float` vs `int`: a number that has a **decimal point** is a `float`, otherwise, it is an `int`
- Lastly, the `bool` data type stores a boolean value which should be either `True` or `False` (note that `T` and `F` are upper-case):

```
is_child: float = False
```

- A summary is found below:

| Type | Min Value | Max Value | Comment |
|---|---|---|---|
| `int` | $-2^{63}-1$ | $2^{63}-1$ | $2^{63} = 9,223,372,036,854,775,808$ (integers only) |
| `float` | 2.22e−308 | 1.79e308 | Supports 300-digit **decimal** value |
| `str` | Supports text of zero or more characters (`""` is zero length) | | |
| `bool` | Supports only two values: `True` or `False` | | |

**Table:** The basic data types in Python

# Commenting Python Code

- It is important for developers to write comments in the code to further explain what is it doing. Comments are **not** executed by the program but there for human beings to understand what is going on. To write a comment, use the hashtag symbol (#) followed by your comment:

```python
import turtle
# The first comment
print("Hello world!") # This line will print Hello World! on console

t = turtle.Turtle()

t.forward(100)
turtle.done()
```

  Some notes:
  - The rest of the line will be considered as a comment not code
  - Comments must be on the same line that has a (#)
  - Having a multi-line comment only works if the comment has a (#)

# Importing Modules

- Python is the most used programming language because of its simple syntax and the sheer amount of available modules
- A *module* is a collection of code that is written once and then shared with people to use
- To use a module, we import it first by placing the import statement at the *very* top of the file
- For example, the following code imports the math module from the standard library[1]:

  ```
  import math
  ```

  And use the functions it contains, such as the square root function:

  ```
  print(math.sqrt(64)) # prints 8.0
  ```

[1] The standard library is a collection of code that was written by Python developers and everyone who uses Python has access to it

# The Turtle Library

| Function | **Description** (assuming the turtle is referred to as `t`) |
|---|---|
| `forward(x)` | Moves forward `x` units<br>**Example:** `t.forward(100)` |
| `backward(x)` | Moves backwards `x` units<br>**Example:** `t.forward(100)` |
| `right(x)` | Turns `x` degrees to the right (always starts pointing → which is 0°)<br>**Example:** `t.right(180)` |
| `left(x)` | Turns `x` degrees to the left (always starts pointing → which is 0°)<br>**Example:** `t.left(45)` |
| `goto(xval, yval)` | Sets the `x` and `y` coordinate as `xval` and `yval`, respectively, and moves to new location<br>**Example:** `t.goto(50, 100)` |
| `setx(val)` | Sets the `x` coordinate value as `val` and moves to new location<br>**Example:** `t.setx(200)` |
| `sety(val)` | Sets the `y` coordinate value as `val` and moves to new location<br>**Example:** `t.sety(150)` |
| `xcor()` | Returns the current `x` coordinate value of the turtle as a `float`<br>**Example:** `t.xcor()` or `print(t.xcor())` to see the value |
| `ycor()` | Returns the current `y` coordinate value of the turtle as a `float`<br>**Example:** `t.ycor()` or `print(t.ycor())` to see the value |
| `speed()` | Returns the speed of the turtle as an `int` and the default value is `3`<br>**Example:** `t.speed()` or `print(t.speed())` to see the value |

# The Turtle Library

| **Function** | **Description** (assuming the turtle is referred to as `t`) |
|---|---|
| `speed(s)` | Sets the speed of the turtle as `s`. Use `3` for slow, `6` for normal, `10` for fast and `0` for super fast. The default value is `3`.<br>**Example:** `t.speed(6)` |
| `heading()` | Returns the current angle as a `float`<br>**Example:** `t.heading()` or `print(t.heading())` to see the value |
| `home()` | Moves turtle to the middle of screen (*i.e.*, `(0, 0)`) facing to the right<br>**Example:** `t.home()` |
| `cirle(r)` | Draws a circle of radius `r`<br>**Example:** `t.circle(16)` |
| `showturtle()` | Shows the turtle pen (*i.e.*, ➤) on the screen (which is the default)<br>**Example:** `t.showturtle()` |
| `hideturtle()` | Hides the turtle pen (*i.e.*, ➤) off the screen<br>**Example:** `t.hideturtle()` |
| `pendown()` | Sets the pen down to draw (which is the default)<br>**Example:** `t.pendown()` |
| `penup()` | Ensures the pen is up so that the turtle wouldn't draw anything<br>**Example:** `t.penup()` |
| `pensize()` | Return the current pen size as an `int` (1 by default)<br>**Example:** `t.pensize()` or `print(t.pensize())` to see the value |
| `pensize(x)` | Sets the pen width as `x`<br>**Example:** `t.pensize(3)` |

# The Turtle Library

| | |
|---|---|
| `pencolor(c)` | Sets the pen color to the specified colour `c`, which is a `str`<br>**Example:** `t.pencolor("red")` |
| `fillcolor(c)` | Sets the fill color to the specified colour `c`, which is a `str`. It shows the effect when used with `t.begin_fill()` and `t.end_fill()`<br>**Example:** `t.fillcolor("magenta")` |
| `color(p, f)` | Sets the pen colour as `p`, the fill color as `f` and both `p` and `f` are `str`s<br>**Example:** `t.color("blue", "yellow")` |
| `begin_fill()` | Ensures to turn on the colour-filling mechanism (don't forget to add `end_fill()` after completing the drawings of the figure)<br>**Example:** `t.end_fill()` |
| `end_fill()` | Ensures to turn off the colour-filling mechanism (don't forget to add `start_fill()` before completing the drawings of the figure)<br>**Example:** `t.end_fill()` |
| `reset()` | Clears all the drawings and brings the turtle to `(0, 0)` pointing $\rightarrow$<br>**Example:** `t.reset()` |
| `turtle.bgcolor(c)` | Sets the background to the specified colour `c`, which is a `str`.<br>**Note:** this is applied on `turtle`, *not* `t`.<br>**Example:** `turtle.bgcolor("red")` |
| `turtle.done()` | Allows the window to stay open after completing the drawings. This **MUST** be placed at the *very* end of the file. **Note:** this is applied on `turtle`, *not* `t`.<br>**Example:** `turtle.done()` |
| `turtle.title(s)` | Changes the title at the top-left of the screen to s, which is a `str`. **Note:** this is applied on `turtle`, *not* `t`.<br>**Example:** `turtle.title("APCO 1P00 Assign 1")` |

# Colour names to use

- In the previous slides, colours were used in functions such as `pencolor`, `fillcolor` and `pencolor`. However, we never mentioned the available colours. Here are the common available colours:

| | | | | | |
|---|---|---|---|---|---|
| `bisque` | `black` | `blueviolet` | `blue` | `brown` | `cadetblue` |
| `chocolate` | `coral` | `crimson` | `cyan` | `darkgray` | `darkgreen` |
| `darkkhaki` | `deeppink1` | `dodgerblue` | `forestgreen` | `gold` | `gray` |
| `green1` | `green2` | `green3` | `green` | `indigo` | `ivory1` |
| `ivory2` | `ivory3` | `lavender` | `lightcoral` | `lightgray` | `lightpink` |
| `limegreen` | `linen` | `magenta2` | `magenta3` | `magenta` | `maroon1` |
| `maroon2` | `maroon3` | `maroon` | `navyblue` | `navy` | `olive` |
| `orange` | `peru` | `pink` | `plum` | `purple` | `red` |
| `silver` | `teal` | `turquoise` | `violet` | `white` | `yellow` |

**Figure:** Common available Python colours

# The Math Library

The Math library/module provides us with all necessary mathematical constants and functions. Here is a list of the popular constants/functions:

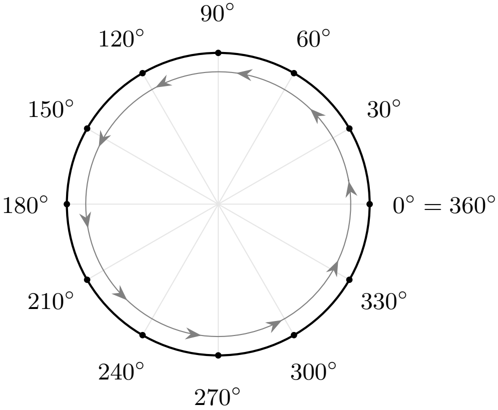
| Term | **Description** (place the example given here inside `print(...)`) |
|---|---|
| `e` | The mathematical constant $e = 2.718281828459045$ |
| `pi` | The mathematical constant $\pi = 3.141592653589793$ |
| `pow(x, y)` | Returns `x` raised to the power `y` (available without the library)<br>**Example:** `pow(7, 2)`<br>**Output :** `49` |
| `ceil(x)` | Returns the ceiling of `x`, the smallest integer greater than or equal to `x`<br>**Example:** `ceil(3.1)`<br>**Output :** `4` |
| `floor(x)` | Returns the floor of `x`, the largest integer less than or equal to `x`<br>**Example:** `floor(4.9)`<br>**Output :** `4` |
| `trunc(x)` | Returns `x` with the fractional part removed, leaving the integer part (similar to `floor` but behaves differently with negative numbers)<br>**Example:** `trunc(5.99999999999)`<br>**Output :** `5` |

# The Math Library

| | |
|---|---|
| `abs(x)` | Returns the absolute value of `x` (available without the library)<br>**Example:** `abs(-6)`<br>**Output** : `6` |
| `fabs(x)` | Returns the absolute value of `x` as a `float`<br>**Example:** `fabs(-7)`<br>**Output** : `7.0` |
| `cos(x)` | Returns the cosine of `x` radians<br>**Example:** `cos(-pi)`<br>**Output** : `-1.0` |
| `sin(x)` | Returns the sine of `x` radians<br>**Example:** `sin(pi / 2)`<br>**Output** : `1.0` |
| `tan(x)` | Returns the tangent of `x` radians<br>**Example:** `tan(pi / 4)`<br>**Output** : `0.9999999999999999` (or 1, occurred due to rounding errors) |
| `degrees(x)` | Convert angle `x` from radians to degrees<br>**Example:** `degrees(pi/2)`<br>**Output** : `90.0` |
| `radians(x)` | Convert angle `x` from degrees to radians<br>**Example:** `radians(180)`<br>**Output** : `3.141592653589793` |

# Understanding Rotations

- Rotations in Turtle Graphics can be based on degrees or radians
- Degrees are values between $0°$ and $359°$ ($360°$ is the same as $0°$, a full rotation) and radians are between 0 and $2\pi$ or 6.28318531
- By default, the degrees convention is used but can be changed to radian
- The degrees are specified based on the following representation (when facing to the right but using `t.left(...)`):



# Complete Turtle Graphics Example Code I

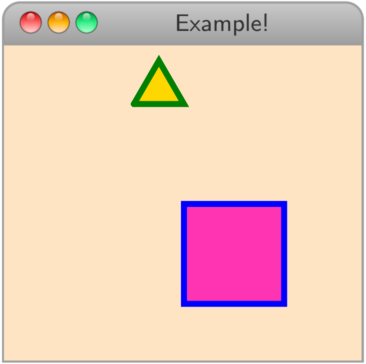
```
import turtle

t = turtle.Turtle()
square_length: int = 50
triangle_length: int = 25
turtle.title("Example!")    # sets the window title
turtle.bgcolor("bisque")    # set the background colour
print(t.pensize())          # prints 1
t.pensize(3)                # now the pen is thicker
print(t.speed())            # prints 3
t.speed(0)                  # super fast!
t.pencolor("blue")
t.fillcolor("maroon1")
# Instead of the two lines above, use the shortcut
# t.color("blue", "maroon1")
t.hideturtle()              # turtle not visible any more
# start drawing a square
t.begin_fill()              # ensures to fill the shape
t.forward(square_length)
print(t.pos())              # prints (50.00,0.00)
t.right(90)
t.forward(square_length)
t.right(90)
t.forward(square_length)
t.right(90)
t.forward(square_length)
t.right(90)
t.end_fill()                # stops shape filling
t.penup()                   # nothing will be drawn
```

# Complete Turtle Graphics Example Code II

```
# Moving to a different location
t.setx(-25)
t.sety(50)
# Instead of the two lines above, use the shortcut
# t.goto(-25, 50)
t.color("green", "gold")    # changing the colours
t.pendown()                 # starts drawing again
# start drawing an equilateral triangle
t.begin_fill()
t.forward(triangle_length)
t.left(120)
t.forward(triangle_length)
t.left(120)
t.forward(triangle_length)
t.left(120)
t.end_fill()
t.showturtle()              # turtle is visible

turtle.done()               # keeps the output visible
```



# Incrementing a variable

- Suppose we have
  ```
  x: int = 7
  ```
- How can we programmatically increment `x` so that it stores the value `8`?
- The syntax (*i.e.*, the way to write it in code) is:
  ```
  x = x + 1
  ```
  which is read as "`x` becomes the current value of `x` plus `1`"
- To decrement the value of `x` by `1`:
  ```
  x = x - 1
  ```
- To multiply the value of `x` by `5`:
  ```
  x = x * 5
  ```
- To divide the value of `x` by `5` using integer division[2]:
  ```
  x = x // 5
  ```

[2] takes the whole number and ignores the decimal points. it doesn't round up nor down, it ignores the decimal points.

# Mathematical Operations

- When dealing with numerical data (`int` or `float`), we are expected to use mathematical operators such as addition, subtraction, etc.
- The order of precedence (*i.e.*, PEDMAS) of Python operations is found below:

| Operator | Symbol | Priority | Example | Output |
|---|---|---|---|---|
| Parentheses | `( )` | highest | `a: int = (5 + 3) * 2` | `16` |
| Exponent | `**` | high | `b: int = 2 ** 10` | `1024` |
| Negation | `-` | high | `c: int = -1` | `-1` |
| Multiplication | `*` | medium | `d: int = 3 * 7` | `21` |
| Division | `/` | medium | `e: float = 16 / 8` | `2.0` |
| Integer Division | `//` | medium | `f: int = 8 // 3` | `2` |
| Modulus | `%` | medium | `g: int = 7 % 2` | `1` |
| Addition | `+` | low | `h: int = 10 + 9` | `19` |
| Subtraction | `-` | low | `i: int = 2 - 7` | `-5` |

**Table:** The order of precedence of Python operators

# Mathematical Operations

- Integer division is performing normal division but ignoring the decimal values and taking only the whole integer
  - For example, `3 / 2 = 1.5` but `3 // 2 = 1` because the `.5` was removed
  - Another example, `10 / 3 = 3.333333333333333` but `10 // 3 = 3`, as the decimal value `.333333333333333` is ignored
- Modulus (also called "mod") is the *remainder* of $x/y$
  - For example: `7 % 2` is $\cancel{2 \times 3}$ *remainder* 1 (we don't care about $2 \times 3$, only the remainder)
  - `8 % 5` is $\cancel{5 \times 1}$ *remainder* 3
  - `10 % 2` is $\cancel{2 \times 5}$ *remainder* 0
  - `75 % 100` is $\cancel{100 \times 0}$ *remainder* 75
  - `1 % 2` is $\cancel{2 \times 0}$ *remainder* 1
  - `2 % 2` is $\cancel{2 \times 1}$ *remainder* 0
  - `3 % 2` is $\cancel{2 \times 1}$ *remainder* 1

# Casting

- *Casting* is the idea of changing the type of a value/variable, for example:
  - Consider the variable, `x: str = "1.0"`, which stores the text `"1.0"`
  - It is currently seen as text, but can we see it as the decimal value `1.0`? How about the integer `1`?
- Casting allows us to make Python see a value or variable as a different type
- Using the example above, we can use
  - `float(x)` to change the text `"1.0"` to a decimal number `1.0`
  - `int(x)` to change the text `"1.0"` to a whole number `1`
  - We can place the text directly without the usage of a variable: `float("1.0")` or `int("1.0")`
- Overall, we can convert from `str` to `float` or `int` and vice-versa as well as convert from `float` to `int` and vice-versa
- Examples are found on the next slide

# Casting

```
int("1")      # converted from text "1" to integer 1
float("2.0")  # converted from text "2.0" to float 2.0
int(3.0)      # converted from a decimal number 3.0 to whole integer 3
float(4)      # converted from a whole integer 4 to decimal 4.0 (i.e., added .0)
str(5)        # converted from a whole integer 5 to text "5"
str(6.0)      # converted from a decimal number 6.0 to text "6.0"
bool("True")  # converted from text "True" to boolean value True
```

- By default, Python will treat values as strings so expect to frequently cast from `str` to your desired type when reading from a file, user input, etc. (this is later on in the course)

**Finding the Type**

To find the type of a variable or value, use `type(...)`. For example, `print(type(5))` will give `<class 'int'>`, concluding that `5` is of the data type `int`.

# Comparison Operators

- Python provides a way to compare quantities using the operations
- They are used for mathematical comparisons and string comparison

| **Operator** | **Definition** | **Example** | **Output** |
|---|---|---|---|
| `<` | strictly less than | `print(1.0 < 2)` | `True` |
| `<=` | less than or equal | `print(3 <= 4)` | `True` |
| `>` | strictly greater than | `print(5 > 6.0)` | `False` |
| `>=` | greater than or equal | `print(7 >= 8.8)` | `False` |
| `==` | equal | `print(9 == 10)` | `False` |
| `!=` | not equal | `print(11.0 != 12)` | `True` |

**Table:** The comparison operators in Python

# Printing on the same line

## Printing on Screen

We use `print("...")` to display something on screen. It will display each print statement on a *new* line. To change this and print on the same line, we could also use commas to write the outputs on the same line like so:

```
print("Hey!", "Bye")
```

but this will eventually add a newline at the end

To ensure all of the output is written on the *same* line, we add `end=" "` to the end of the print statement:

```
print("Hey!", end=" ")
```

We specified a space character in `end=" "` so that a space is added instead of a newline. We could write the following to add a comma instead

```
print("Hey!", end=",")
```

# Single For-loop

- In Python, a `for` loop is a way for us to repeat an action $x$ amount of times
- We need to declare a variable (usually named `i`) and use the `range` function
- The `range` function has three values: starting index, ending index and an increment value (the increment value **must** be an `int`):
  `range(starting_index, ending_index, increment)`
- The `ending_index` is **NOT** included in the range
- Here is an example where we start at `0`, end at `10` (but not including `10`) and increment by `1` each iteration to loop 10 times in total:

```
for i in range(0, 10, 1):
  print(i)
# output: 0 1 2 3 4 5 6 7 8 9
```

- Notes to consider:
  – The `print` statement is indented into the `for` loop (this is a *must*!)
  – The `range` functions starts at `0` (and including `0`) but goes to less than `10`
  – There is a colon (*i.e.*, `:`) at the end of the `for` loop
  – Not indenting the `print` statement yields in the following error:
  `IndentationError: expected an indented block after 'for' statement`

# Single for-loop

- Here are other examples:

```
for i in range(3, 14, 2):
  print(i, end=" ")
# output: 3 5 7 9 11 13
print(" ")
for i in range(-10, 2, 1):
  print(i, end=" ")
# output: -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1
print(" ")
for i in range(10, 3, -1):
  print(i, end=" ")
# output: 10 9 8 7 6 5 4
```

- There are some shortcuts that can be applied on this:

  ```
  for i in range(0, 10, 1): # loops through 0 1 2 3 4 5 6 7 8 9
  ```

  - In case the increment is `1`, then only specify the starting and ending values, *i.e.*,

    ```
    for i in range(0, 10): # loops through 0 1 2 3 4 5 6 7 8 9
    ```

  - In case the starting value is `0` and increment is `1`, only specify the ending value *i.e.*,

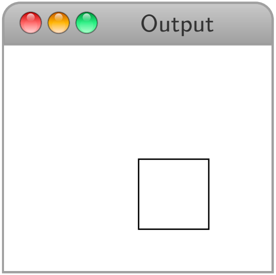
    ```
    for i in range(10): # loops through 0 1 2 3 4 5 6 7 8 9
    ```

# Single for-loop

- We can use a for loop to draw a square instead of having to write the same code multiple times:

```
import turtle

t = turtle.Turtle()
for i in range(4): # i = 0, 1, 2, 3
    t.forward(50)
    t.right(90)
turtle.done()
```




- Note that both of the commands are indented which means they are a part of the for loop
- The line `turtle.done()` is *not* a part of the loop, which means it will run only once

# Single for-loop

**Quick Exercise**

Using two `for` loops, draw a square of length `50`, then, use the `goto` function to move some other place, then draw another square of length `100`.

- In case we need to create a `for` loop that will loop but wouldn't print anything, we cannot just leave it blank because Python will complain about indentation. Instead, we use the `pass` keyword which does nothing, like so:

```
for i in range(4):
  pass # do nothing statement
```

# Nested for-loops

- Nested loops are defined when a `for` loop is inside another `for` loop
- Typically, we use the variable `i` in the outer loop and variable `j` in the inner one
- Here is an example where the outer loop loops `3` times and the inner one loops `4` times:

```
for i in range(3):       # outer loop, i = 0, 1, 2
  for j in range(4):     # inner loop, j = 0, 1, 2, 3
    pass # do nothing
```

  – Note how the inner loop is indented and the `pass` statement is indented in the inner loop
  – The logic is as follows:
    - Start at `i = 0`, now run the inner loop which loops `4` times (`j` goes from `0` to `3`)
    - Loop back up and now `i = 1`, run the inner loop (which runs `4` times in total, `j` goes from `0` to `3`)
    - Loop back up and now `i = 2`, run inner loop (`j` goes from `0` to `3`)
    - We now have completed all iterations as the valid values of `i` are `0`, `1` and `2`

# Nested for-loops

- Consider the following example:

```
for i in range(2):   # outer loop, i = 0, 1
  for j in range(3): # inner loop, j = 0, 1, 2
    print("i = ", i, end=", ") # print the value of i
    print("j = ", j)           # print the value of j
```

Output

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
```

From the output, `i` changes twice and `j` has three values and completes two iterations because `i` looped twice

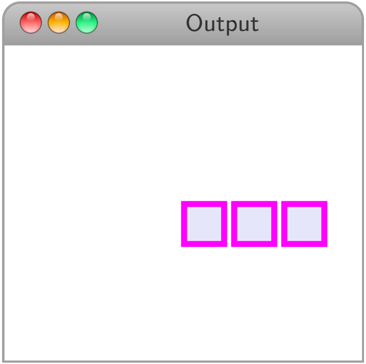
# Nested loops

- We could repeat the same task over and over again using nested loops and Turtle Graphics:

```
import turtle

t = turtle.Turtle()
t.pensize(3)
t.color("magenta", "lavender")
for i in range(3):    # draws three squares in total
  t.pendown()
  t.begin_fill()
  for j in range(4):  # draws a single filled square
    t.forward(20)
    t.right(90)
  t.end_fill()
  t.penup()  # don't draw anything
  # forward to the same length (20) and another
  # 5 units to place a divider
  t.forward(25)
turtle.done()
```

# Nested loops




- Python allows us to have nested loops or three-level nesting or even more

# Nested vs Non-nested for-Loops

- Non-nested `for` loops are loops that will execute in order and the number of iterations is added. For example:

  ```
  for i in range(12): # first loop
    print(i)


  for j in range(5):  # second loop
    print(j)
  ```

  In total, there will be $12 + 5 = 17$ print statements
- Nested `for` loops are loops that will execute in a mixed order and the number of iterations is multiplied. For example:

  ```
  for i in range(12):   # outer loop
    for j in range(5):  # inner loop
      print(i, ", ", j)
  ```

  In total, there will be $12 \times 5 = 60$ print statements

# Nested vs Non-nested for-Loops

- A mix of nested and non-nested loops also follows the same rule, for example:

```
for i in range(10):   # outer loop
  for j in range(20): # first inner loop
    print(j)
  for k in range(50): # second inner loop
    print(k)
```

  - There will be 700 print statements in total
  - The first inner loop and second inner loop a total of 70 iterations
  - Since the outer loop loops 10 times and the total inner loops is 70, the overall total is $10 \times 70 = 700$

# while-loops

- `while` loops are a different way to achieve repetition
- It contains a condition that must be `True` to continue looping
- Once the condition is `False`, the looping stops
- Used when the number of iterations is unknown
  - A real-life example is when asking the user to enter their email and password. We don't know how many times they will get it wrong (*i.e.*, they can get it correct on their first try or 8th one)
- Example: `i` is initialized to `0` and the `while` loop will loop `10` times (from 0 to 9):

```python
i: int = 0
while (i < 10):
  print(i, end=" ")
  i = i + 1  # increment i by 1
# output: 0 1 2 3 4 5 6 7 8 9
```

  - As long as `i` is `0`, `1`, ..., `8` or `9`, the condition will be `True` because these values are less than `10`
  - When `i` becomes `10`, then the condition `10 < 10` is not `True`, hence, it becomes `False` and then stops

# Infinite loops

- We are able to create loops that never stop looping by using a `while` loop that has a condition of `True`
- Since the condition is always true and the loop will be valid at all times:

```
while True:
  print("Hey")
# output: keeps on printing Hey on screen
```

# Conditional Statements

- Conditional statements are commands for specifying which code to run in the case where the requirement is satisfied
- A `while` loop did have a condition and executed the code as long as it is `True`
- In here, we are not dealing with loops but with just conditional statements.
- We check the condition once and then execute code based on whether it is true or not
- No looping involved!

# if-statement

- Python also has an `if` statement where it executes code *only once* if the condition is `True`, for example:

```
if 1 < 5:
  print("One is less than five")
# output: One is less than five
```

  - The condition is that `1` must be less than `5` to run the code in the `if` statement; Since this is the case, it printed out the text
  - The code inside the `if` statement executed only once! It is not a loop so it cannot execute more than once
- Consider this next example:

```
if 0 > 1:
  print("zero is larger than one")
# nothing displayed as output
```

  - We didn't enter the code inside the `if` statement because the condition `0 > 1` is not `True` (*i.e.*, `False`)
  - Hence, the code inside the `if` statement executed zero times

# if-else statement

- Suppose that we are interested in both branches of the condition, *i.e.*, the `True` and the `False` outcomes
- We can use an `if` and `else` statements to cover both branches:

```
if 2 < 3:
  print("The condition is true")
else:
  print("The condition is false")
# output: The condition is true
```

  - Since the condition is `True`, *only* the code in the `if` branch was executed
- Another example:

```
if 6 < 5:
  print("Six less than five")
else:
  print("Six is not less than five")
# output: Six is not less than five
```

  - Since the condition is `False`, *only* the code in the `else` branch was executed