

# BRANCHING, ITERATION

---

6.0001 LECTURE 2

# TODAY

---

- string object type
- branching and conditionals
- indentation
- iteration and loops

# STRINGS

---

- letters, special characters, spaces, digits
- enclose in **quotation marks or single quotes**

```
hi = "hello there"  
"123"
```

- **concatenate** strings (addition)

```
name = "ana"
```

```
greet = hi + name
```

```
greeting = hi + " " + name
```

- **Repetition operation**

```
silly = name * 3 = name + name + name
```

## Errors

- Error 1: **NameError**: name 'a' is not defined

```
>>> a
```

Because a is not a literal of any type, the interpreter treats it as a name. But that name is not bound to any object, attempting to use it causes a runtime

- Error 2: **TypeError**: can't multiply sequence by non-int of type 'str'

```
>>> 'a'*'a'
```

# Other operations can be used in str type

## 1. Length of a String:

- Use `len()` to find the length of a string
  - Ex: `len('abc')` returns 3.

## 2. Indexing:

- To get individual characters from a string using indexing, which starts at 0
  - Ex: `'abc'[0]` gives 'a'
- Negative indices start from the end of the string.
  - Ex: `'abc'[-1]` returns 'c'

## 3. Slicing:

- To get a part of the string.
- Syntax `s[start:stop]` which returns the substring from `start` to `stop-1`
  - Ex: `'abc'[1:3]` returns 'bc'
- Omitting the start value defaults it to 0, and omitting the end value defaults it to the string's length.
  - `'abc'[:]` is the same as `'abc'[0:3]`

# INPUT/OUTPUT: `print`

---

- used to **output** stuff to console
- keyword is `print`

```
x = 1
```

```
print(x)
```

```
x_str = str(x)
```

```
print("my fav num is", x, ".", "x =", x)
```

```
print("my fav num is " + x_str + ". " + "x = " + x_str)
```

# INPUT/OUTPUT: `input ("")`

---

- prints whatever is in the quotes
- user types in something and hits enter
- binds that value to a variable

```
text = input("Type anything... ")  
print(5*text)
```

- `input` **gives you a string** so must cast if working with numbers

```
num = int(input("Type a number... "))  
print(5*num)
```

# COMPARISON OPERATORS ON `int`, `float`, `string`

---

- `i` and `j` are variable names
- comparisons below evaluate to a Boolean

`i > j`

`i >= j`

`i < j`

`i <= j`

used to compared 2 strings ex: `'a' > 'ab'`  
or 2 numbers only ex: `3 > 2`

`i == j` → **equality** test, True if `i` is the same as `j`

`i != j` → **inequality** test, True if `i` not the same as `j`

# LOGIC OPERATORS ON bools

---

- `a` and `b` are variable names (with Boolean values)

**`not a`**     $\rightarrow$    True if `a` is False  
                     False if `a` is True

**`a and b`**  $\rightarrow$    True if both are True

**`a or b`**     $\rightarrow$    True if either or both are True

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False



# COMPARISON EXAMPLE

---

```
pset_time = 15  
sleep_time = 8  
print(sleep_time > pset_time)  
derive = True  
drink = False  
both = drink and derive  
print(both)
```

# CONTROL FLOW – BRANCHING-conditional statement

```
if <condition>:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
elif <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

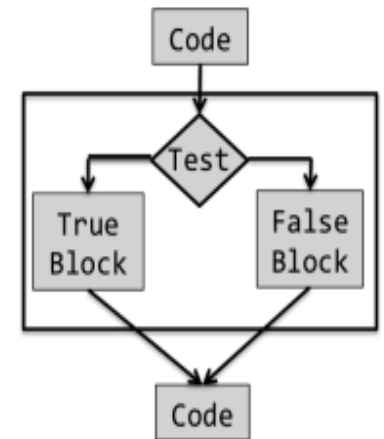


Figure 2.3 Flow chart for conditional statement

- `<condition>` has a value True or False
- evaluate expressions in that block if `<condition>` is True

*If  $x < y$  and  $x < z$  → compound Boolean expression*

# INDENTATION

---

- matters in Python
- how you denote blocks of code

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")
print("thanks!")
```

nested ←

# = VS ==

---

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")
print("thanks!")
```

What if  $x = y$  here?  
get a `SyntaxError`

# CONTROL FLOW: Iteration

## while LOOPS

```
while <condition>:  
    <expression>  
    <expression>  
    ...
```

- <condition> evaluates to a Boolean
- if <condition> is True, do all the steps inside the while code block
- check <condition> again
- repeat until <condition> is False

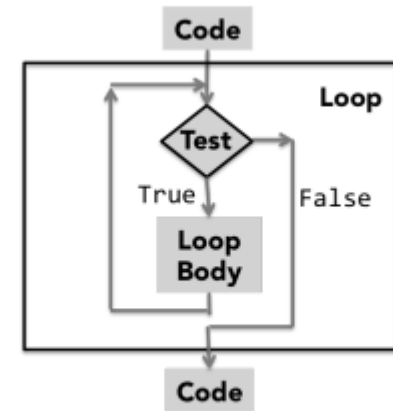


Figure 2.4 Flow chart for iteration

# CONTROL FLOW: `for` LOOPS

```
for <variable> in range(<some_num>) :  
    <expression>  
    <expression>  
    ...
```

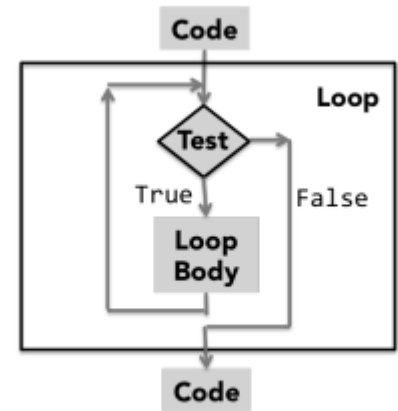


Figure 2.4 Flow chart for iteration

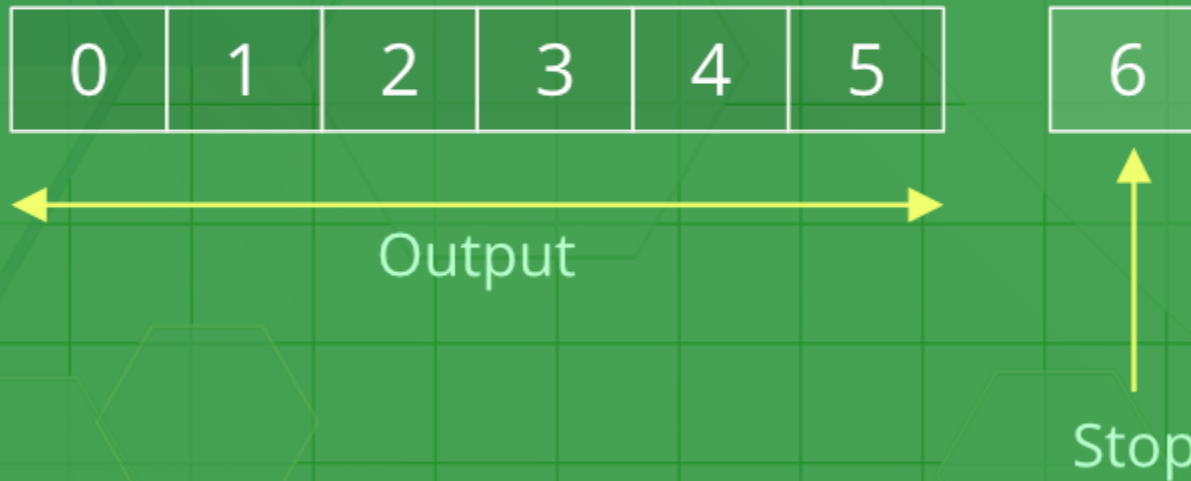
- each time through the loop, `<variable>` takes a value
- first time, `<variable>` starts at the smallest value
- next time, `<variable>` gets the prev value + 1
- etc.

# range (start, stop, step)

- *start: [ optional ] start value of the sequence. Default value = 0*
- *stop: next value after the end value of the sequence. loop until value is **stop - 1***
- *step: [ optional ] integer value, denoting the difference between any two numbers in the sequence*

## Python range (stop)

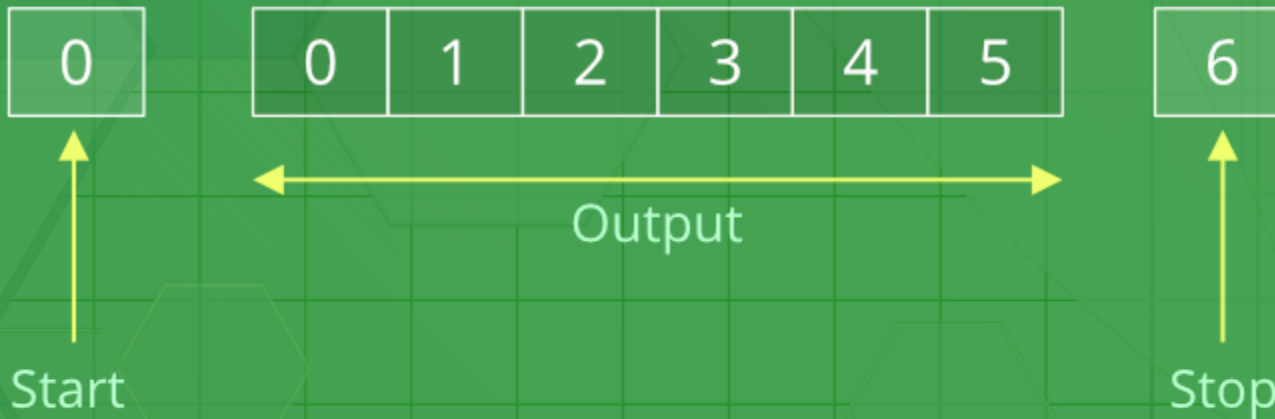
Python range( 6 )





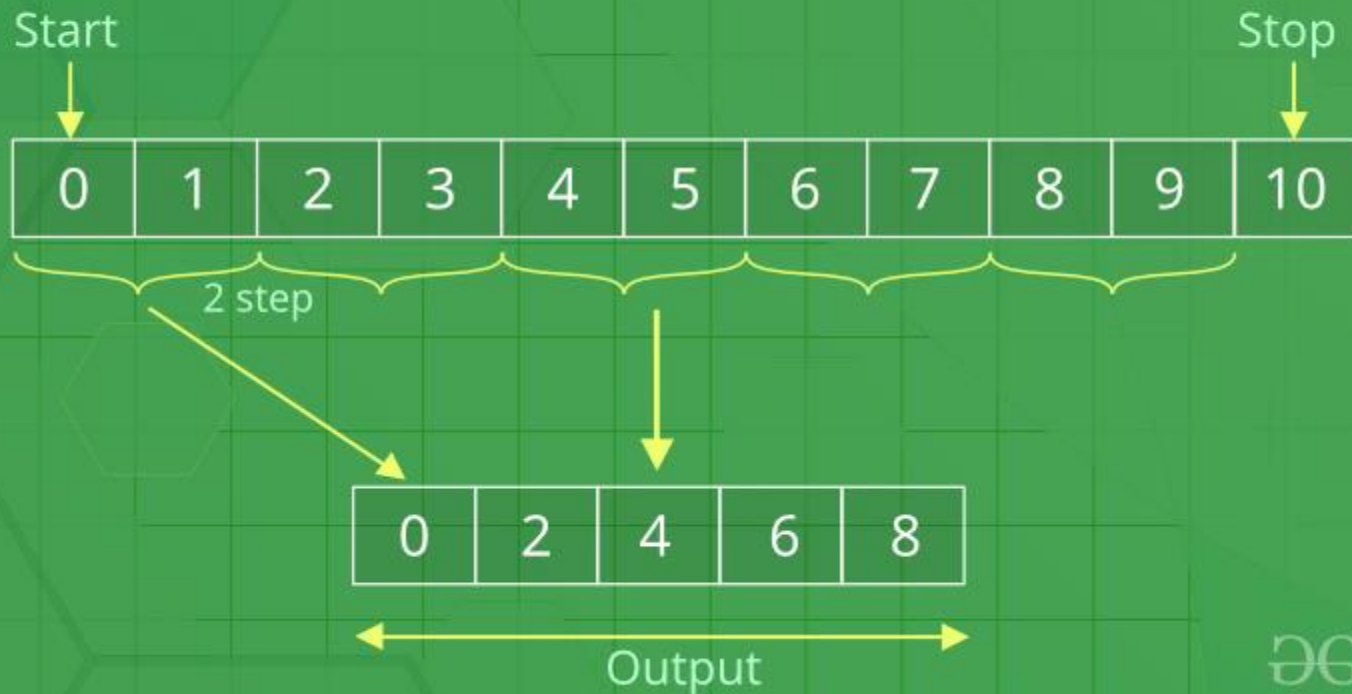
## Python range (start, stop)

Python range( 0, 6 )



## Python range (start, stop, step)

Python range( 0, 10, 2 )



## Important points to remember about range() function

- The range() function only works with integers, i.e. whole numbers.
- All arguments must be integers. Users can not pass a string or float number or any other type in a **start**, **stop**, and **step** argument of a range().
- All three arguments can be positive or negative.
- The **step** value must not be zero. If a step is zero, python raises a ValueError exception.
- Users can access items in a range() by index, just as users do with a list. (Explain in next page)

### 1. Accessing elements in range by index:

```
r = range(5, 15) # This creates a range object with numbers from 5 to 14  
print(r[0]) # Output: 5  
print(r[4]) # Output: 9  
print(r[-1]) # Output: 14
```

### 2. Iterating over elements in a range object:

```
r = range(3, 8) # This creates a range object with numbers from 3 to 7  
for i in range(len(r)):  
    print(r[i])
```

#Output: 3 4 5 6 7

### 3. Slicing a range object:

```
r = range(20, 30) # This creates a range object with numbers from 20 to 29  
sliced_range = r[2:5]  
print(list(sliced_range)) # Output: [22, 23, 24]
```

# break STATEMENT

---

- immediately exits whatever loop it is in
- skips remaining expressions in code block
- exits only innermost loop!

```
while <condition_1>:  
    while <condition_2>:  
        <expression_a>  
        break  
        <expression_b>  
    <expression_c>
```

# break STATEMENT

```
mysum = 0
for i in range(5, 11, 2):
    mysum
    f mysum == 5
        break
    mysum += 1
print(mysum)
```

- what happens in this program?

# for VS while LOOPS

---

## for loops

- **know** number of iterations
- can **end early** via `break`
- uses a **counter**
- **can rewrite** a `for` loop using a `while` loop

## while loops

- **unbounded** number of iterations
- can **end early** via `break`
- can use a **counter but must initialize** before loop and increment it inside loop
- **may not be able to rewrite** a `while` loop using a `for` loop

# while and for LOOPS

## examples

---

- iterate through numbers in a sequence

```
# more complicated with while loop
n = 0
while n < 5:
    print(n)
    n = n+1
```

```
# shortcut with for loop
for n in range(5):
    print(n)
```