Lecture 3: Strings and I/O

Comp 102

Forman Christian University

Recap

Strings

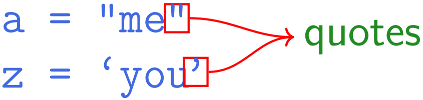
Strings

`string` is a sequence of **case sensitive** characters

```
a = "me"
z = ‘you’
```

Strings

`string` is a sequence of **case sensitive** characters

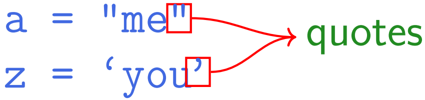


```
a = "me"
z = 'you'
```

quotes

Strings

string is a sequence of **case sensitive** characters

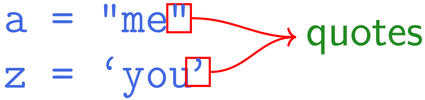


```
a = "me"
z = ‘you’
```

Concatenate and **Repeat** strings:

Strings

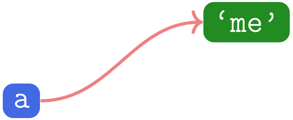
string is a sequence of **case sensitive** characters



```
a = "me"
z = ‘you’
```

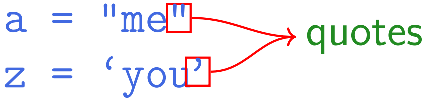
quotes

Concatenate and **Repeat** strings:



Strings

`string` is a sequence of **case sensitive** characters



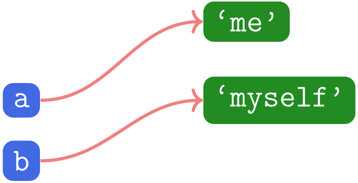
```
a = "me"
z = ‘you’
```

quotes

Concatenate and **Repeat**

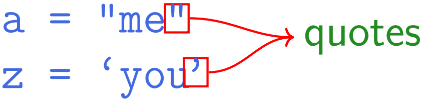
strings:

➡ `b = "myself"`



Strings

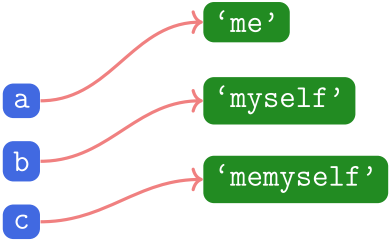
`string` is a sequence of **case sensitive** characters



```
a = "me"
z = 'you'
```

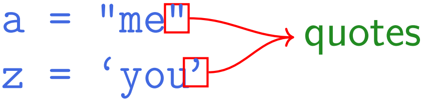
Concatenate and **Repeat** strings:

```
b = "myself"
c = a + b
```



Strings

`string` is a sequence of **case sensitive** characters

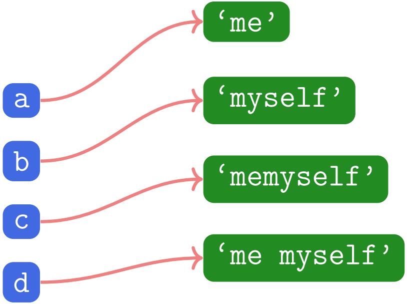


```
a = "me"
z = 'you'
```

Concatenate and **Repeat**

strings:

```
b = "myself"
c = a + b
d = a + " " + b
```



Strings

`string` is a sequence of **case sensitive** characters

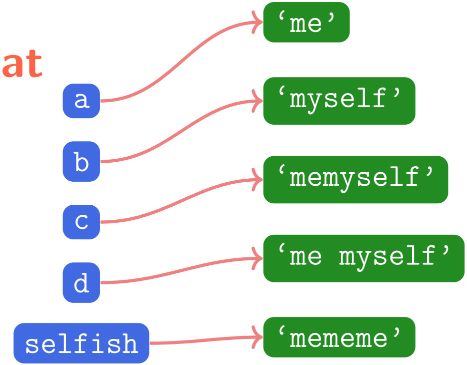
```
a = "me"
z = ‘you’
```

quotes

Concatenate and **Repeat**

strings:

```
b = "myself"
c = a + b
d = a + " " + b
selfish = a * 3
```



a → ‘me’

b → ‘myself’

c → ‘memyself’

d → ‘me myself’

selfish → ‘mememe’

You Try

What's the value of `s1` and `s2`?

-
  ```
  b = ":"
  c = ")"
  s1 = b + 2*c
  ```

-
  ```
  f = "a"
  g = "b"
  h = "3"
  s2 = (f + g) * int(h)
  ```

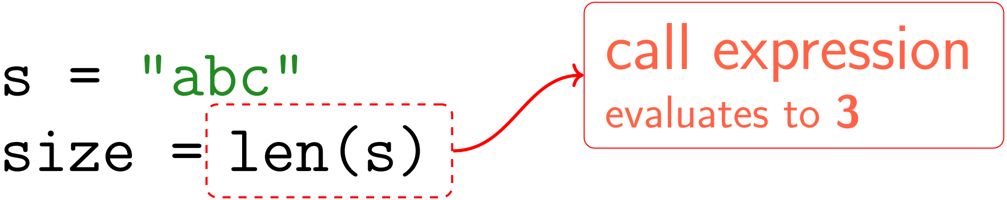
String Length

`len()` returns the length of a string, **excluding** quotes.

```
s = "abc"
size = len(s)
```

String Length

`len()` returns the length of a string, **excluding** quotes.



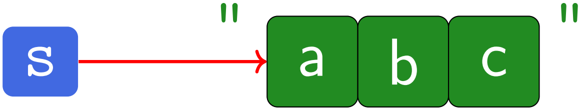
```
s = "abc"
size = len(s)
```

call expression

evaluates to **3**

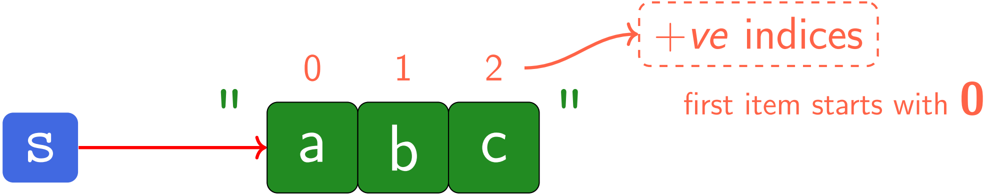
Getting a **single** **character** from a string

Every character in a string has an **index** *(a.k.a position)*.



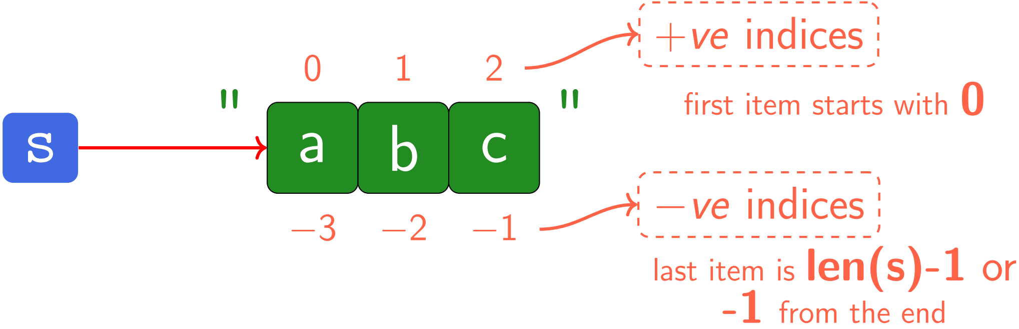
Getting a **single** **character** from a string

Every character in a string has an **index** (*a.k.a position*).



Getting a **single character** from a string

Every character in a string has an **index** (*a.k.a* *position*).



s
"
a
b
c
"
0
1
2
−3
−2
−1
+*ve* indices
first item starts with **0**
−*ve* indices
last item is **len(s)-1** or **-1** from the end

You Try

What's index of the following characters in this string:

```
s = "comp102"
```

- `c`
- `1`
- `2`
- `m`
- `p`
- `0`

You Try

What's index of the following characters in this string:

`s = "comp102"`

- `c` → **0 or -7**
- `1` → **4 or -3**
- `2` → **6 or -1**
- `m`
- `p`
- `0`

You Try

What's index of the following characters in this string:

```
s = "comp102"
```

- `c` → **0 or -7**
- `1` → **4 or -3**
- `2` → **6 or -1**
- `m` → **2 or -5**
- `p` → **3 or -4**
- `0` → **5 or -2**

Once you know the index, you can **read** the character from the string using [] square brackets.

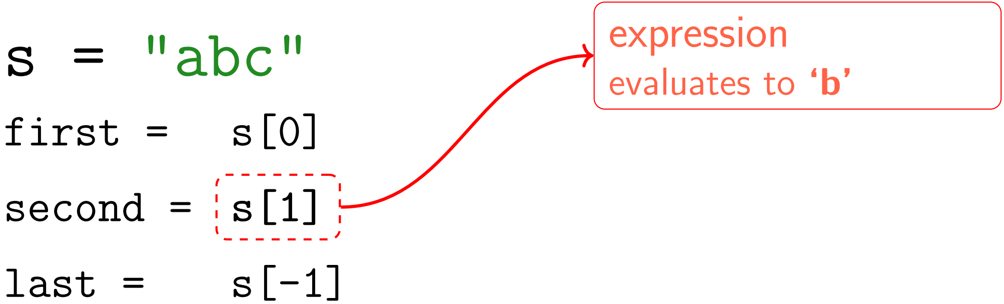
```
s = "abc"
first =   s[0]
second =  s[1]
last =    s[-1]
```

Once you know the index, you can **read** the character from the string using [] square brackets.

```
s = "abc"
first =   s[0]
second =  s[1]
last =    s[-1]
```

expression
evaluates to **‘a’**

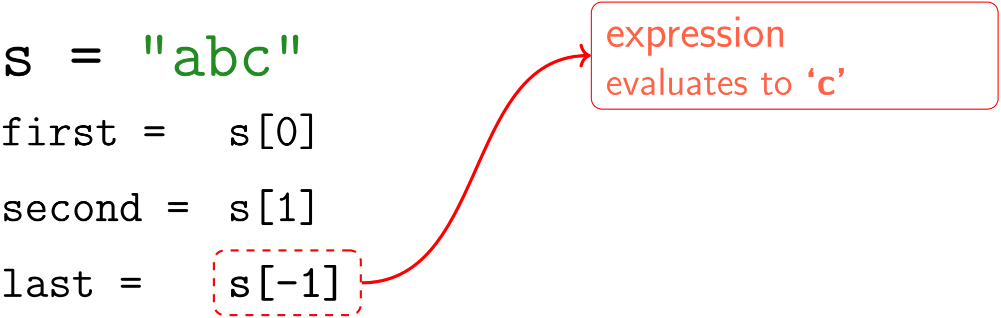
Once you know the index, you can **read** the character from the string using [] square brackets.



```
s = "abc"
first =   s[0]
second =  s[1]
last =    s[-1]
```

expression
evaluates to **‘b’**

Once you know the index, you can **read** the character from the string using [] square brackets.



```
s = "abc"
first =   s[0]
second =  s[1]
last =    s[-1]
```

expression evaluates to **'c'**

Once you know the index, you can **read** the character from the string using [] square brackets.

```
s = "abc"
first =   s[0]
second =  s[1]
last =    s[-1]
out =     s[3]
```

trying to index out of bounds, **ERROR**

Slicing to get a Substring

Slicing to get a Substring

- To **Slice** means to get a part of a string

Slicing to get a Substring

- To **Slice** means to get a part of a string
- You can slice strings using `[start:stop:step]`
  - Get characters at indices `start`
    up to and including `stop-1`
    taking `step` step characters

Slicing to get a **Substring**

- To **Slice** means to get a part of a string
- You can slice strings using `[start:stop:step]`
  - Get characters at indices `start`
    up to and including `stop-1`
    taking `step` step characters
- If give two numbers, `[start:stop]`, `step=1` by default

Slicing to get a Substring

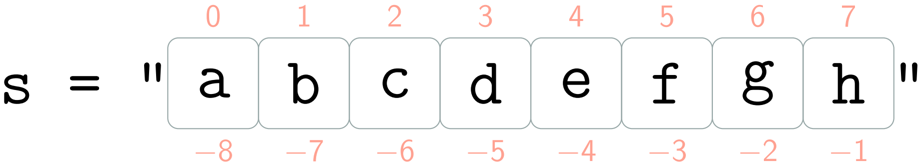
- To **Slice** means to get a part of a string
- You can slice strings using `[start:stop:step]`
  - Get characters at indices `start`
    up to and including `stop-1`
    taking `step` step characters
- If give two numbers, `[start:stop]`, `step=1` by default
- If give one number, `[num]`, you're back to indexing for a single character *(prev slide)*

Slicing to get a Substring

- To **Slice** means to get a part of a string
- You can slice strings using `[start:stop:step]`
  - Get characters at indices `start`
    up to and including `stop-1`
    taking `step` step characters
- If give two numbers, `[start:stop]`, `step=1` by default
- If give one number, `[num]`, you're back to indexing for a single character *(prev slide)*
- You can also omit numbers and leave just colons *(try out in terminal!)*

Slicing Examples

- You can slice strings using `[start:stop:step]`
- Look at the `step` first.
  +*ve* means go left to right,
  −*ve* means go right to left

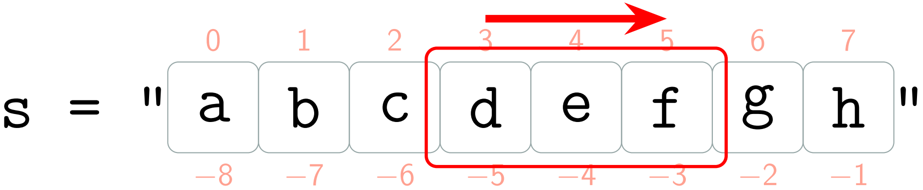


s = "abcdefgh"

	0	1	2	3	4	5	6	7
s =	a	b	c	d	e	f	g	h
	−8	−7	−6	−5	−4	−3	−2	−1

Slicing Examples

- You can slice strings using `[start:stop:step]`
- Look at the `step` first.
  +*ve* means go left to right,
  −*ve* means go right to left



0 1 2 3 4 5 6 7

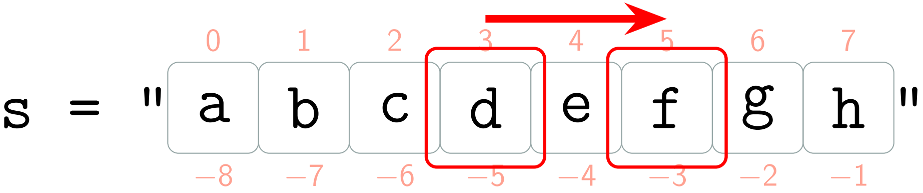
s = "a b c d e f g h"

−8 −7 −6 −5 −4 −3 −2 −1

`s[3:6]` → evaluates to `"def"`, same as `s[3:6:1]`

Slicing Examples

- You can slice strings using `[start:stop:step]`
- Look at the `step` first.
  +*ve* means go left to right,
  −*ve* means go right to left



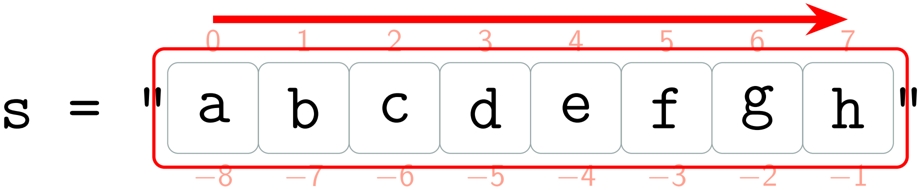
s = "abcdefgh"

	0	1	2	3	4	5	6	7
s = "	a	b	c	d	e	f	g	h
	−8	−7	−6	−5	−4	−3	−2	−1

`s[3:6:2]` → evaluates to `"df"`

Slicing Examples

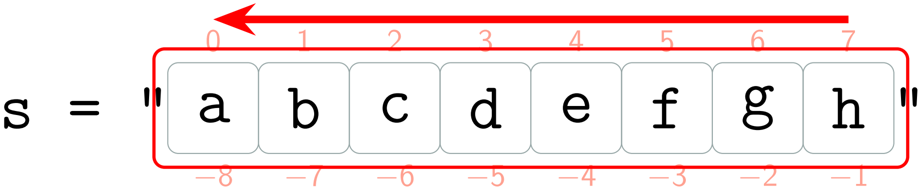
- You can slice strings using `[start:stop:step]`
- Look at the `step` first.
  +*ve* means go left to right,
  −*ve* means go right to left



`s[:]` → evaluates to `"abcdefgh"`, same as `s[0:len(s):1]`

Slicing Examples

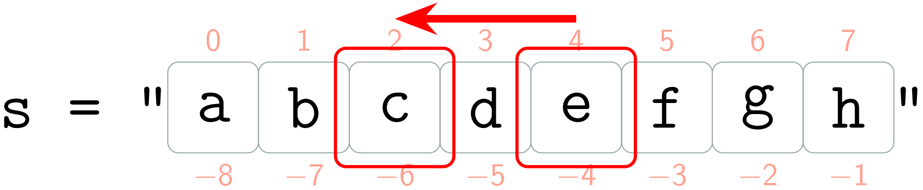
- You can slice strings using `[start:stop:step]`
- Look at the `step` first.
  +*ve* means go left to right,
  −*ve* means go right to left



`s[::-1]` → evaluates to `"hgfedcba"`

Slicing Examples

- You can slice strings using `[start:stop:step]`
- Look at the `step` first.
  $+ve$ means go left to right,
  $-ve$ means go right to left



`s[4:1:-2]` → evaluates to `"ec"`

You Try

```
s = "ABC d3f ghi"
s[3:len(s)-1]
s[4:0:-1]
s[6:3]
```

Strings are IMMUTABLE

- **Immutable** means you can’t change a string

```
s = "car"
char = s[2]    ✓
s[0] = ‘b’    ✗,   ERROR
```

Strings are IMMUTABLE

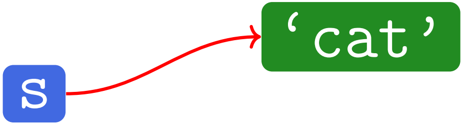
- **Immutable** means you can't change a string
- However, you can create **new** strings from existing ones

```
s = "cat"
s = ‘r’ + s[1:len(s)]
```

Strings are IMMUTABLE

- **Immutable** means you can't change a string
- However, you can create **new** strings from existing ones

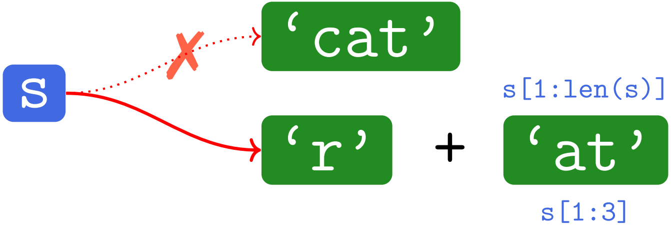
```
s = "cat"
s = ‘r’ + s[1:len(s)]
```



Strings are IMMUTABLE

- **Immutable** means you can't change a string
- However, you can create **new** strings from existing ones

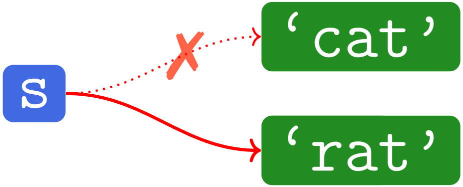
```
s = "cat"
s = ‘r’ + s[1:len(s)]
```



Strings are IMMUTABLE

- **Immutable** means you can't change a string
- However, you can create **new** strings from existing ones

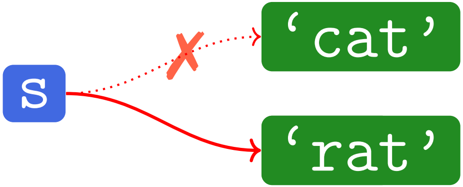
```
s = "cat"
s = ‘r’ + s[1:len(s)]
```



Strings are IMMUTABLE

- **Immutable** means you can't change a string
- However, you can create **new** strings from existing ones

```
s = "cat"
s = ‘r’ + s[1:len(s)]
```



Remember: a variable binds to exactly ONE value

Big Idea

If you are wondering *“what happens if”*...

Just try it out in the console!

Input and Output (I/O)

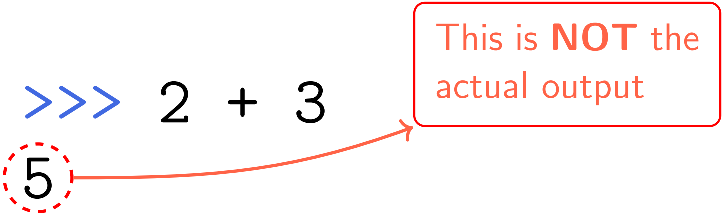
Interpreter Output

Typing an expression in the interpreter window:

```
>>> 2 + 3
5
```

Interpreter Output

Typing an expression in the interpreter window:

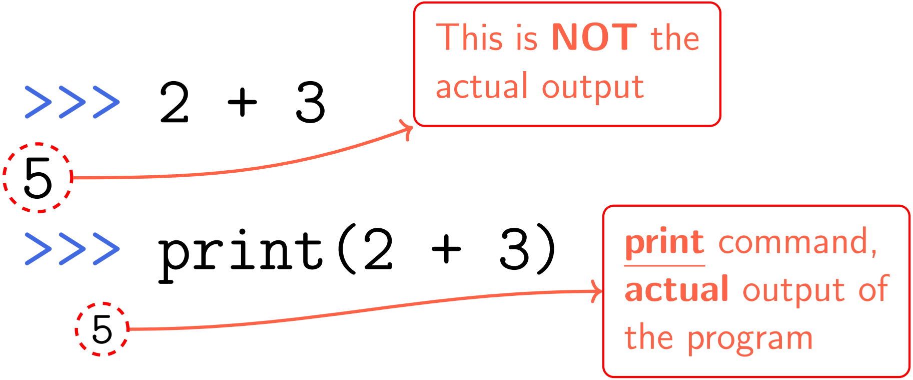


```
>>> 2 + 3
5
```

This is **NOT** the actual output

Interpreter Output

Typing an expression in the interpreter window:



```
>>> 2 + 3
5
>>> print(2 + 3)
5
```

This is **NOT** the actual output

print command, **actual** output of the program

Big Idea(s)

1. The only way a program prints on the screen is by using the `print()` command
2. When program gets larger, save in a **file**

You Try

Write the following statements in the code editor section:

```
print("hi")
print(2 + 3)
2 + 3
"hi"*3
```

What do you see in the output window when you run this code?

The print Command

```
a = "The"

b = 7

c = "warriors"
```

The print Command

```
a = "The"

b = 7

c = "warriors"

print(a, b, c)
```

Separate objects using **commas** to output them separated by **spaces**

The print Command

```
a = "The"
b = 7
c = "warriors"
print(a, b, c)
print(a + str(b) + c)
```

Concatenate strings together using + to print as single object

The print Command

```
a = "The"

b = 7

c = "warriors"

print(a, b, c)

print(a + str(b) + c)
```

Every piece being concatenated must be a **string**

You Try

Write the following statements in the code editor section:

```
a = "The"
b = 7
c = "warriors"
print(a, b, c)
print(a + str(b) + c)
```

What do you see in the output window when you run this code?

You Try

Identify the error(s) and fix the following code:

```
a = 5
b = 3.14
c = "pi"
s1 = "The value of a is " + a
s2 = "The value of b is " + b
s3 = "The value of c is " + c
```

Input

```
x = input(s)
```

1. Prints the value of the string `s`
2. User types in something and hits enter
3. That value is assigned to the variable `x`

Input

```
x = input("Type anything: ")
print(3 * x)
```

Shell

```
Type anything:
```

and it waits for characters and **Enter** *key to be hit*

Input

```
x = input( "Type anything:  " )
print(3 * x)
```

Shell

```
Type anything: hello
```

Input

```
x = input( "Type anything:  " )
print(3 * x)
```

Hitting enter will create a **string** object in the memory

`'hello'`

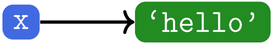
Shell

```
Type anything:  hello ↵
```

Input

```
x = input( "Type anything:  " )
print(3 * x)
```

then it **Binds** that value to a variable



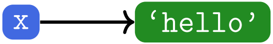
Shell

```
Type anything: hello ↵
```

Input

```
x = input("Type anything:  ")
print(3 * x)
```

then it **Binds** that value to a variable



Shell

```
Type anything:  hello ↵
hellohellohello
```

Input

```
num1 = input("Type a number:  ")
print(5 * num1)
num2 = int( input("Type a number:  ") )
print(5 * num2)
```

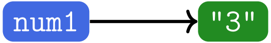
Shell

Input

"3"

```
num1 = input("Type a number:  ")
print(5 * num1)
num2 = int( input("Type a number:  ") )
print(5 * num2)
```

1. input always returns a **string**



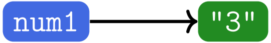
Shell

```
Type a number: 3 ⏎
```

Input

```
num1 = input("Type a number:  ")
print(5 * num1)
num2 = int( input("Type a number:  ") )
print(5 * num2)
```

1. input always returns a **string**



Shell

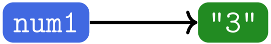
```
Type a number:  3
33333
```

Input

```
num1 = input("Type a number:  ")
print(5 * num1)
num2 = int(input("Type a number:  "))
print(5 * num2)
```

"3"

1. input always returns a **string**



Shell

```
Type a number:  3
33333
Type a number:  3
```

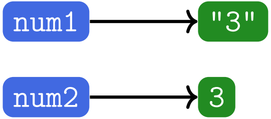
Input

```
num1 = input("Type a number:  ")
print(5 * num1)
num2 = int( input("Type a number:  ") )
print(5 * num2)
```

3

1. input always returns a **string**

2. must **cast** if working with **numbers**



Shell

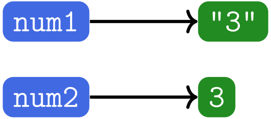
```
Type a number:  3
33333
Type a number:  3
```

Input

```
num1 = input("Type a number:  ")
print(5 * num1)
num2 = int( input("Type a number:  ") )
print(5 * num2)
```

1. input always returns a **string**

2. must **cast** if working with **numbers**



Shell

```
Type a number:  3
33333
Type a number:  3
15
```

exactly same input, completely different output

You Try

Write a program that:

- Asks the user for a verb
- Prints `"I can ___ better than you"`, where ___ is any verb
- Then prints the verb 5 times in a row, separated by spaces
- For example, if the user enters `run`, you print:

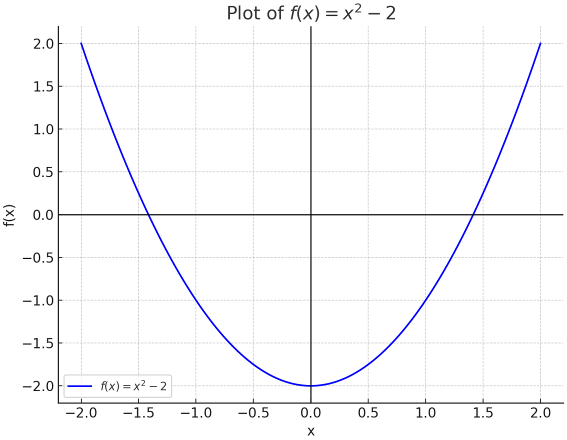
```
I can run better than you!
run run run run run
```

Newton-Raphson Method

(find approximate roots of a function)

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

$f(x) = x^2 - 2$ has roots: $\pm\sqrt{2}$



Newton-Raphson Method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

For $f(x) = x^2 - 2$ and $f'(x) = 2x$, the formula becomes:

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n} = \left(\frac{x_n}{2} + \frac{1}{x_n}\right) = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Square Root *(Approximation)*

Newton-Raphson Method

Let's say we want to find $\sqrt{a}$, Newton's equation for square roots:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Square Root *(Approximation)*

Newton-Raphson Method

Let's say we want to find $\sqrt{a}$, Newton's equation for square roots:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Example: $\sqrt{2}$

- Start with an initial guess: $x_0 = 1$
- Keep on calculating $x_1, x_2, x_3, \ldots$

Square Root *(Approximation)*

Newton-Raphson Method

Let's say we want to find $\sqrt{a}$, Newton's equation for square roots:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Example: $\sqrt{2}$

- Start with an initial guess: $x_0 = 1$
- Keep on calculating $x_1, x_2, x_3, \ldots$

$$\begin{aligned}
x_0 &= \texttt{1.0} \\
x_1 &= \texttt{1.5} \\
x_2 &= \texttt{1.4166666666666665} \\
x_3 &= \texttt{1.4142156862745097} \\
x_4 &= \texttt{1.4142135623746899} \\
x_5 &= \texttt{1.414213562373095}0
\end{aligned}$$

Square Root *(Approximation)*

Newton-Raphson Method

Let’s say we want to find $\sqrt{a}$, Newton’s equation for square roots:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Partial code of algorithm that gets input and finds next guess:

```
# Try Newton Raphson for square root
a = int(input(’What a to find the square root of?  ’))
g = float(input(’What guess to start with?  ’))
next_g = 0.5 * (g + a/g)
print(’Next guess to try =’, next_g)
```

f-strings

- Available starting with Python 3.6
- Character f followed by a **formatted string literal**
  - Anything that can be appear in a normal string literal
  - Expressions bracketed by curly braces { }
- Expressions in curly braces evaluated at runtime, automatically converted to strings, and concatenated to the string preceding them

```
num = 3000
fraction = 1/3
print(num*fraction, 'is', fraction*100, '% of', num)
```

f-strings

- Available starting with Python 3.6
- Character `f` followed by a **formatted string literal**
  - Anything that can be appear in a normal string literal
  - Expressions bracketed by curly braces { }
- Expressions in curly braces evaluated at runtime, automatically converted to strings, and concatenated to the string preceding them

```
num = 3000
fraction = 1/3
print(num*fraction, ’is’, fraction*100, ’% of’, num)
```

introduces an extra space

f-strings

- Available starting with Python 3.6
- Character `f` followed by a **formatted string literal**
    - Anything that can be appear in a normal string literal
    - Expressions bracketed by curly braces { }
- Expressions in curly braces evaluated at runtime, automatically converted to strings, and concatenated to the string preceding them

```
num = 3000
fraction = 1/3
print(num*fraction, ’is’, fraction*100, ’% of’, num)
print(num*fraction, ’is’, str(fraction*100) + ’% of’, num)
```

introduces an extra space

f-strings

- Available starting with Python 3.6
- Character `f` followed by a **formatted string literal**
  - Anything that can be appear in a normal string literal
  - Expressions bracketed by curly braces { }
- Expressions in curly braces evaluated at runtime, automatically converted to strings, and concatenated to the string preceding them

```
num = 3000
fraction = 1/3
print(num*fraction, ’is’, fraction*100, ’% of’, num)
print(num*fraction, ’is’, str(fraction*100) + ’% of’, num)
print(f"{num*fraction} is {fraction*100}% of {num}")
```

introduces an extra space

expressions

Big Idea

Expressions can be placed anywhere.

Python always evaluates them!

You Try

Try the running the following code and see what it prints:

```
sp = 2

print(f"Speed Doubled:2*sp m/s")

print("Speed squared:",sp**2,"m/s")

print(f"Speed cubed:{sp**3}m/s")
```

Practice

You Try

What is the **value** and **type** of each expression?

Expression	Value	Type
`3 + 4`		
`3.0 + 4`		
`"3" + "4"`		
`"ha" * 3`		
`10 // 3`		
`10 / 3`		

You Try

What is the **value** and **type** of each expression?

Expression	Value	Type
`3 + 4`	`7`	`int`
`3.0 + 4`	`7.0`	`float`
`"3" + "4"`		
`"ha" * 3`		
`10 // 3`		
`10 / 3`		

You Try

What is the **value** and **type** of each expression?

Expression	Value	Type
`3 + 4`	`7`	`int`
`3.0 + 4`	`7.0`	`float`
`"3" + "4"`	`"34"`	`str`
`"ha" * 3`	`"hahaha"`	`str`
`10 // 3`		
`10 / 3`		

You Try

What is the **value** and **type** of each expression?

Expression	Value	Type
`3 + 4`	`7`	`int`
`3.0 + 4`	`7.0`	`float`
`"3" + "4"`	`"34"`	`str`
`"ha" * 3`	`"hahaha"`	`str`
`10 // 3`	`3`	`int`
`10 / 3`	`3.333...`	`float`

You Try

What does each expression evaluate to? Or does it **error**?

Expression	Result
`int("42")`	
`float("3")`	
`str(100)`	
`int(7.9)`	
`int("hello")`	
`str(3) + str(4)`	

You Try

What does each expression evaluate to? Or does it **error**?

Expression	**Result**
`int("42")`	42
`float("3")`	3.0
`str(100)`	"100"
`int(7.9)`	
`int("hello")`	
`str(3) + str(4)`	

You Try

What does each expression evaluate to? Or does it **error**?

Expression	**Result**
`int("42")`	`42`
`float("3")`	`3.0`
`str(100)`	`"100"`
`int(7.9)`	7 (truncates, does NOT round!)
`int("hello")`	`ERROR`
`str(3) + str(4)`	

You Try

What does each expression evaluate to? Or does it **error**?

Expression	**Result**
`int("42")`	`42`
`float("3")`	`3.0`
`str(100)`	`"100"`
`int(7.9)`	`7` (truncates, does NOT round!)
`int("hello")`	`ERROR`
`str(3) + str(4)`	`"34"`

You Try

What is the value of c, d, e, and f after this code runs?

```
a = 10
b = 3
c = a / b
d = int(c)
e = str(d) * b
f = len(e)
```

You Try

What is the value of `c`, `d`, `e`, and `f` after this code runs?

```
a = 10
b = 3
c = a / b
d = int(c)
e = str(d) * b
f = len(e)
```

c → 3.333... d → 3 e → "333" f → 3

You Try

What does this program print if the user enters 4?

```
x = input("Number:  ")
print(x * 3)
print(int(x) * 3)
print(f"Result:{int(x) ** 2}")
```

You Try

What does this program print if the user enters 4?

```
x = input("Number:  ")
print(x * 3)
print(int(x) * 3)
print(f"Result:{int(x) ** 2}")
```

444 (string "4" repeated 3 times)

You Try

What does this program print if the user enters 4?

```
x = input("Number:  ")
print(x * 3)
print(int(x) * 3)
print(f"Result:{int(x) ** 2}")
```

444 (string "4" repeated 3 times)

12 (integer 4 multiplied by 3)

You Try

What does this program print if the user enters 4?

```
x = input("Number:  ")
print(x * 3)
print(int(x) * 3)
print(f"Result:{int(x) ** 2}")
```

```
444         (string "4" repeated 3 times)
12          (integer 4 multiplied by 3)
Result:16   (4 squared, inside f-string)
```

You Try

Identify the error(s) and fix the code so it prints: `Area:78.5`

```
r = input("Radius? ")
area = 3.14 * r ** 2
print("Area: " + area)
```

You Try

Identify the error(s) and fix the code so it prints: `Area:78.5`

```
r = input("Radius? ")
area = 3.14 * r ** 2
print("Area: " + area)
```

Error 1: `r` is a **string**. Need `float(r)` or `int(r)`

Error 2: Can't concatenate `str + float`. Need `str(area)` or use f-string

You Try

Identify the error(s) and fix the code so it prints: `Area:78.5`

```
r = input("Radius? ")
area = 3.14 * r ** 2
print("Area: " + area)
```

Fixed:

```
r = int(input("Radius? "))
area = 3.14 * r ** 2
print(f"Area: {area}")
```

You Try

Write a program that:

- Asks the user for their **name** and **birth year**
- Calculates their approximate age in 2025
- Prints: `Hello <name>! You are about <age> years old.`
- Then prints their name **reversed**
- For example, if the user enters `Ali` and 2005:

```
Hello Ali!  You are about 20 years old.
ilA
```

Summary

Summary

- **Types:** `int`, `float`, `str`
- **Conversion:** `int()`, `float()`, `str()`
- **Strings:** concatenate (+), repeat (∗), index (`[]`), slice (`[::]`), `len()`, **immutable**
- **Variables:** bind a name to a value with =
- **Expressions:** Python always evaluates them
- **I/O:** `print()` for output, `input()` for input (always returns a `str`)
- **f-strings:** `f"...{expr}..."` for formatted output

Questions ?