CSCi 1012 [Section 10]

Introduction to Programming with Python

Prof. Kartik Bulusu, CS Dept.

Course start date January 17, 2024
Lecture location 1957 E street Room 213
Lecture times Monday, 3:45 PM to 5:00 PM

Wednesday-lab
3:45 PM to 5:00 PM
Section-30: MON 352
Section-31: SEH 4040
Section-34: TOMP 310
Section-35: TOMP 204

Friday-lab
3:45 PM to 5:00 PM
Section-32: SEH 4040
Section-33: TOMP 309
Section-36: TOMP 306
Section-37: TOMP 107

GW School of Engineering & Applied Science
THE GEORGE WASHINGTON UNIVERSITY

Spring 2024

Photo: Kartik Bulusu

Class Policy on Collaboration

- **You may *not* *discuss Modules, Assignments, Quizzes and Exams* among yourselves.**
- **Each student is expected to work out the course deliverables *independently*.**
- **Under *no circumstances* may you look at another student's *Modules, Assignments, Quizzes and Exams*, or look for answers to *Modules, Assignments, Quizzes and Exams* anywhere other than in the text in the course website.**
- **You are encouraged to discuss the class material on Ed-discussion board or in-person with the instruction team.**
- **You may *not* discuss *Modules, Assignments, Quizzes and Exams* nor give out hints for the same on problems on the Ed-discussion board or with other students in-person.**

All violations will be treated as violations of the Code of Academic Integrity.

HWs
- Due dates
- Late work
- Extensions

Date	Topic(s)	Wednesday Lab Date	Friday Lab Date	Assignment(s)
Week 9 [03/18/2024]	Functions	03/20/2024	03/22/2024	Unit 1 » Module 2 (Due **March 25, 2024** by **11:59 PM**)
Week 13 [04/15/2024]	**Examination**	04/17/2024	04/19/2024	Unit 2 » Module 0 & Module 1 (Due **April 22, 2024** by **11:59 PM**)

- **IMPORTANT:** Please attend the ONLY lab that you registered into.

Late Work

- **Late work is not accepted, with the following exceptions:**
  - Every student many turn in **as many as four (in total, not each) assignments or modules 48 hours after the deadline with no penalty**. Requesting an extension is not necessary.
- **Extensions** will be **granted should there arise circumstances beyond your control** that impede your ability to complete coursework.
  - Notify your professor as soon as feasible in these cases.
    - Examples of such circumstances include (but are not limited to) illness, death in the family, and loss of housing. To ensure fairness toward all students, we will request documentation of such circumstances.

Submission Tips

- Pay attention to file names

extra spaces

assignment1.zip ≠ ***assignment 1.zip***

caesar_shift.py ≠ ***caesar_shift .py***

extra spaces

- Pay attention to small details

```
x is 3 and j is 2, x + j = 5
x is 3 and j is 3, x + j = 6
x is 3 and j is 4, x + j = 7
x is 3 and j is 5, x + j = 8
x is 3 and j is 6, x + j = 9
x is 3 and j is 7, x + j = 10
x is 3 and j is 8, x + j = 11
```

≠

missing commas

```
x is 3 and j is  2 x + j 5
x is 3 and j is  3 x + j 6
x is 3 and j is  4 x + j 7
x is 3 and j is  5 x + j 8
x is 3 and j is  6 x + j 9
x is 3 and j is  7 x + j 10
x is 3 and j is  8 x + j 11
```

extra space *missing =*

- Don't add extra endline spaces

print("++++", end = "") →

```
++++
+  +
+  +
++++
```

→ ***What Auto Grader Sees…***

'++++\n+ +\n+ +\n+ +\n++++'

≠

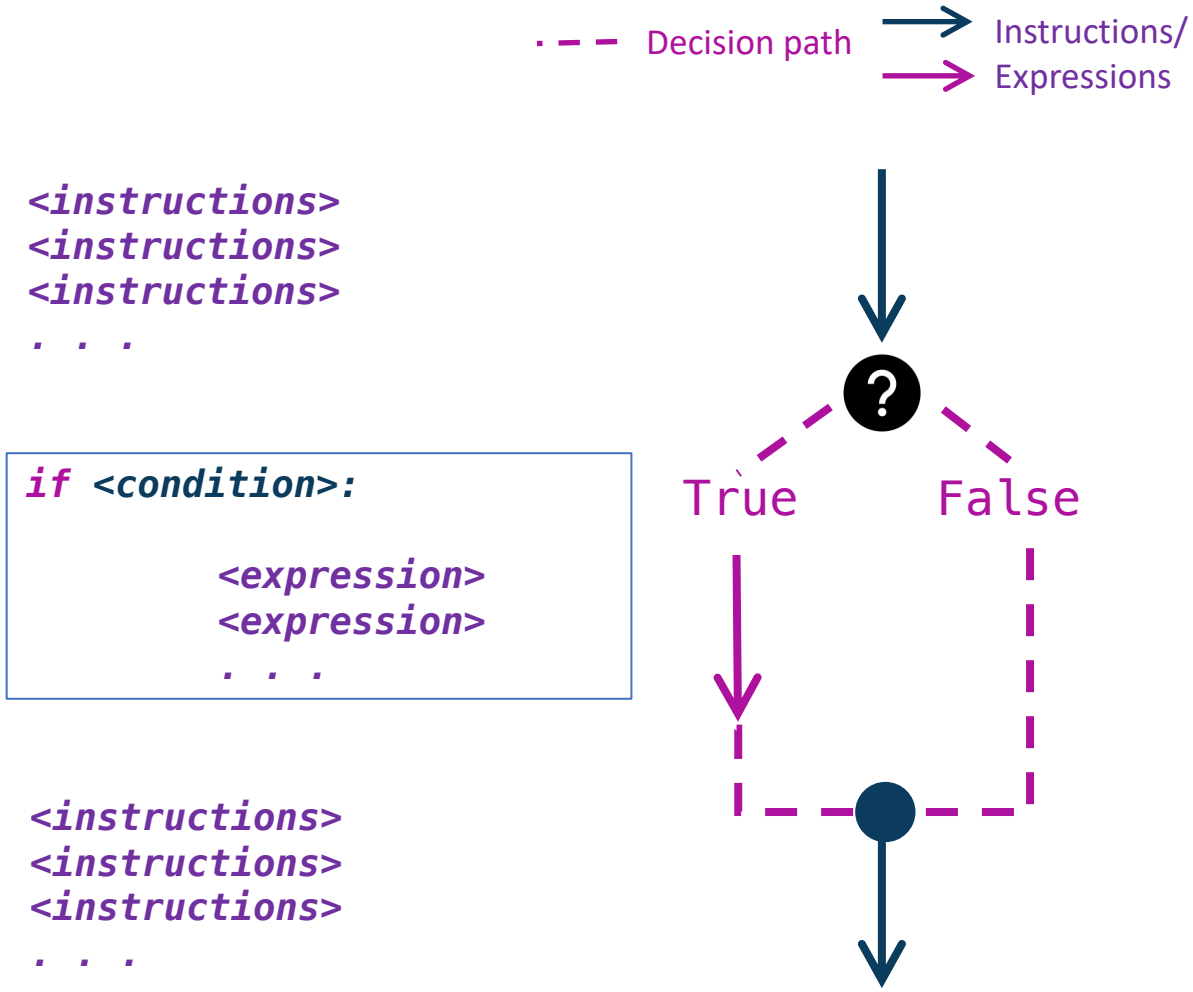
print("++++", end = " ") →

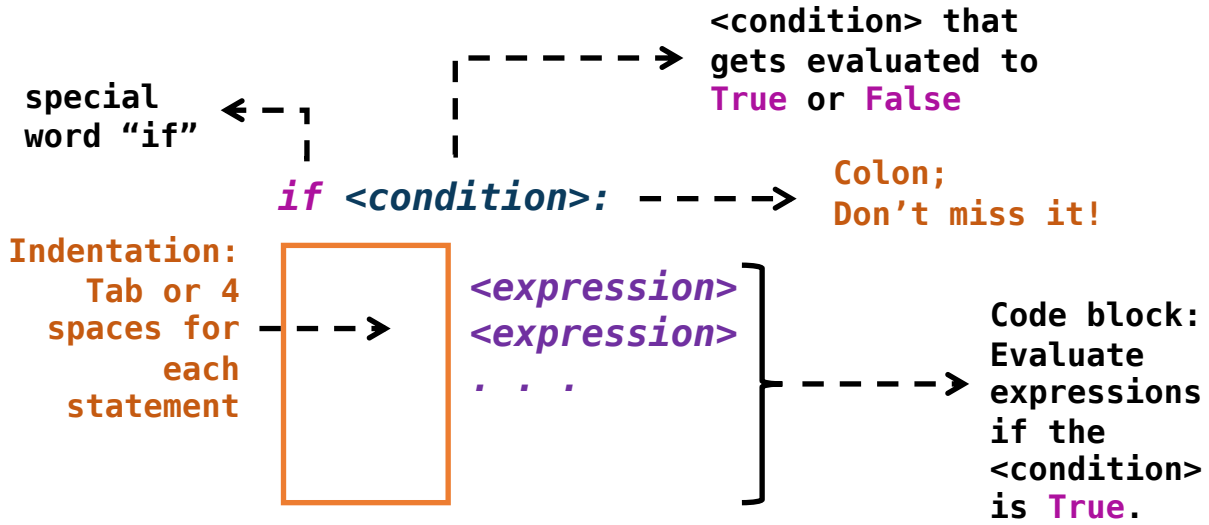
```
++++
+  +
+  +
++++
```

→ '++++ \n+ +\n+ +\n+ +\n++++ '

extra spaces

Skeleton of the control flow - branching using `if-construct`



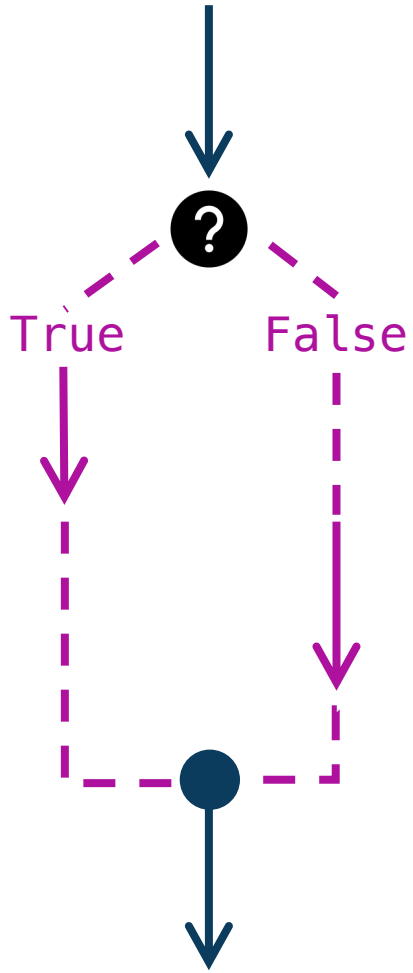


Skeleton of the control flow - branching using `if-else-construct`

Decision path

Instructions/ Expressions

```
<instructions>
<instructions>
. . .

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

<instructions>
<instructions>
. . .
```



special word "if"

<condition> that gets evaluated to True or False

`if <condition>:` → Colon; Don't miss it!

Indentation: Tab or 4 spaces for each statement

`<expression>`
`<expression>`
`. . .`

Code block: Evaluate expressions if the <condition> is True.

`else:` → Colon; Don't miss it!

Indentation: Tab or 4 spaces for each statement

`<expression>`
`<expression>`
`. . .`

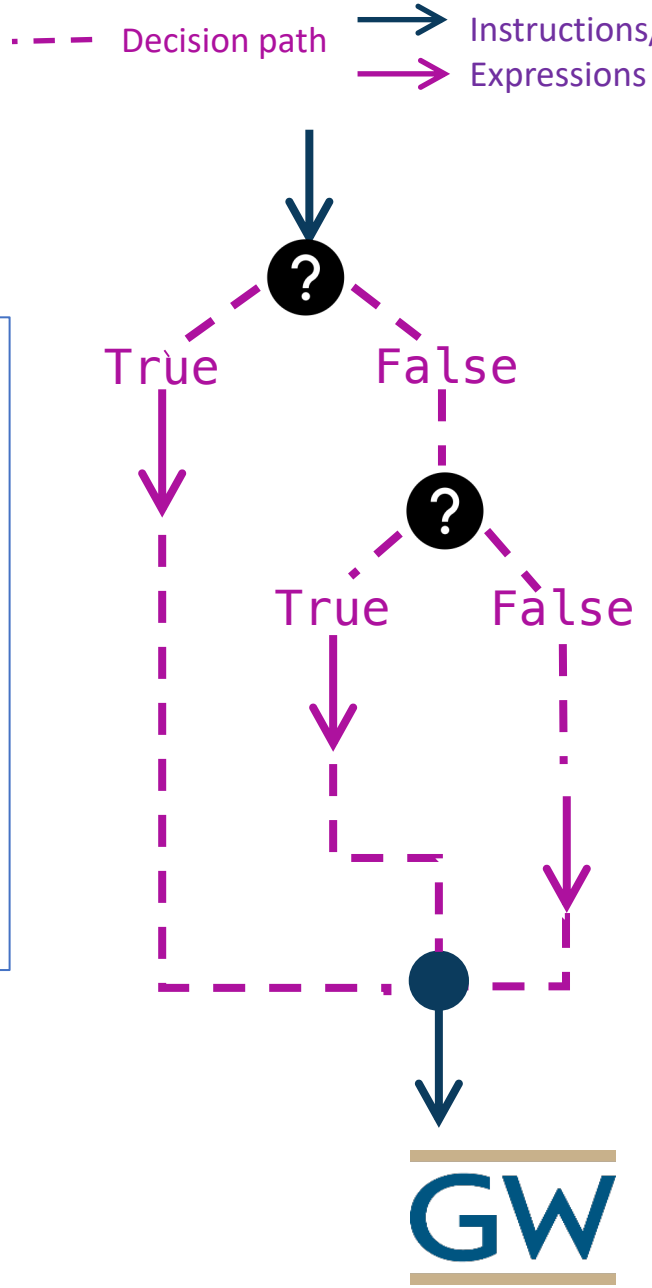
Code block: Evaluate expressions if the preceding <condition> is False.

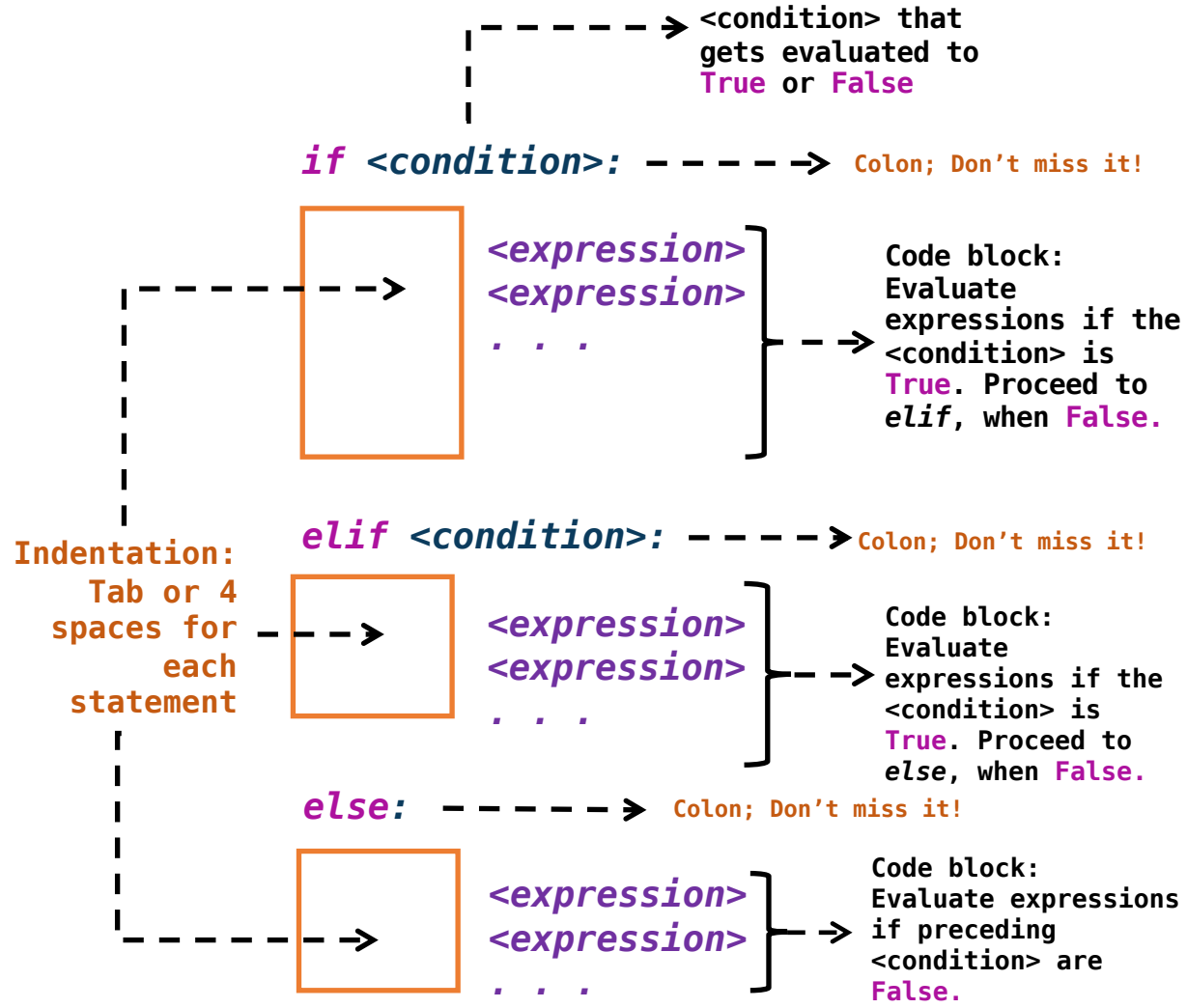
School of Engineering & Applied Science
THE GEORGE WASHINGTON UNIVERSITY

GW

Skeleton of the control flow - branching using `if-elif-else-construct`

```
<instructions>
. . .

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

elif <condition>:
        <expression>
        <expression>
        . . .

else:
        <expression>
        <expression>
        . . .

<instructions>
. . .
```





RECAP: Functions - A first look

A `function` is group of related statements that performs a specific task.

`functions`

- Break your program into smaller and modular chunks.
- Make a program more manageable and modular as a program grows larger and larger
- Avoid repetition and makes the code reusable

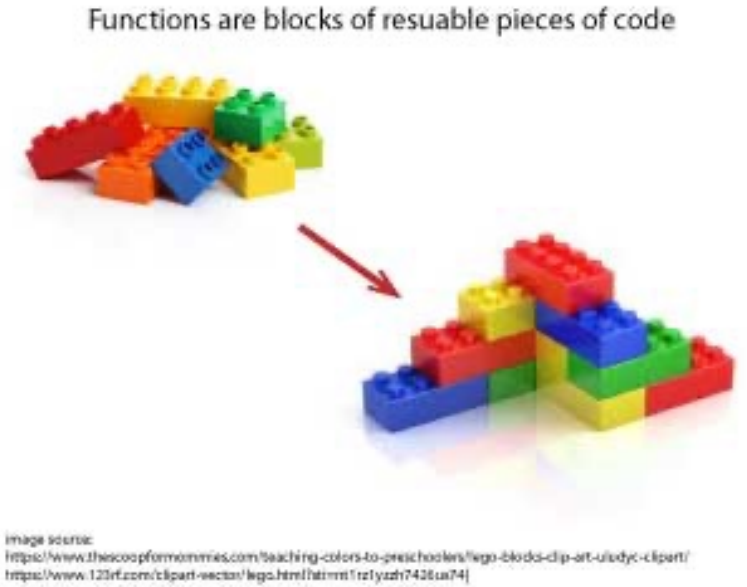


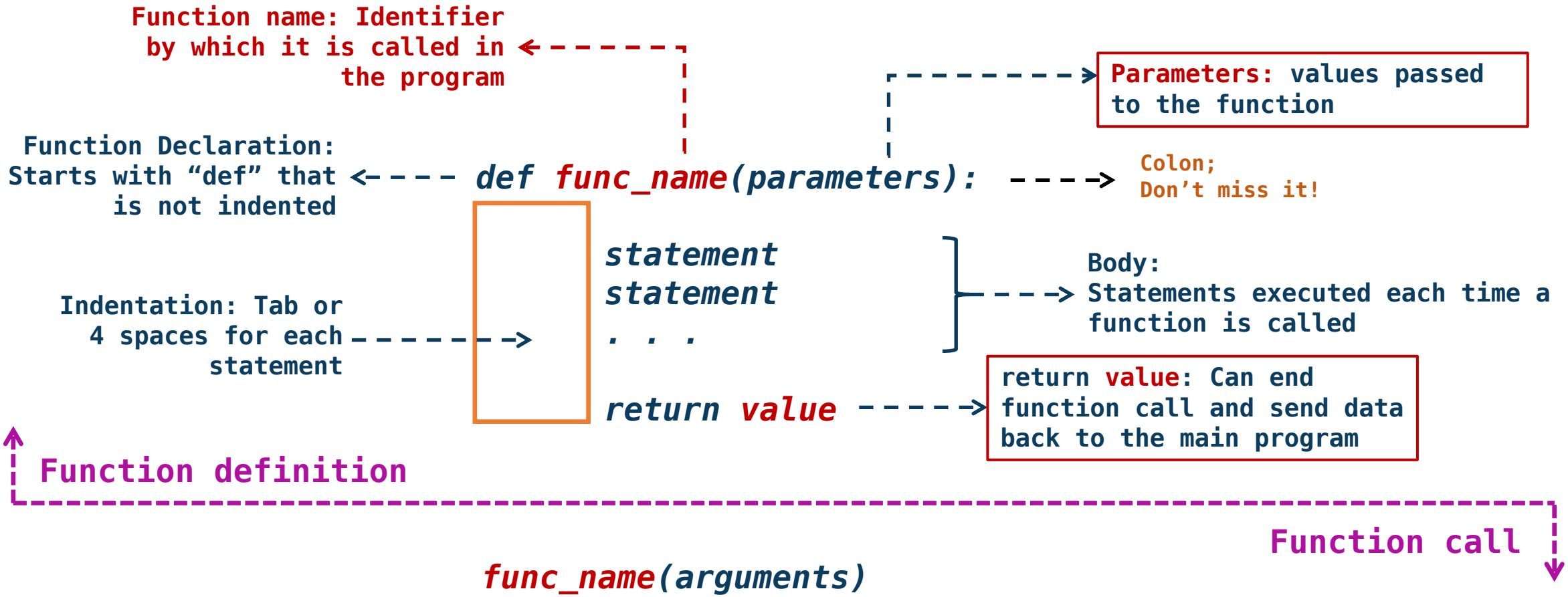
Image source:
https://www.thescoopformommies.com/teaching-colors-to-preschoolers/lego-blocks-clip-art-ulsdyc-clipart/
https://www.123rf.com/clipart-vector/legos.html?sti=mt1rz1yzzh7426ua74|

`print()`
`range()`
`len()`
`input()`

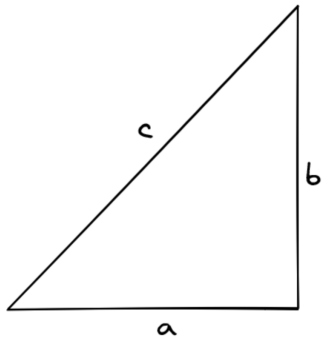
- Example of a function you just learned
- Built-in function in Python

Syntax and Skeleton of a user-defined `function`

User-defined `function` – The second look



Functions with multiple parameters



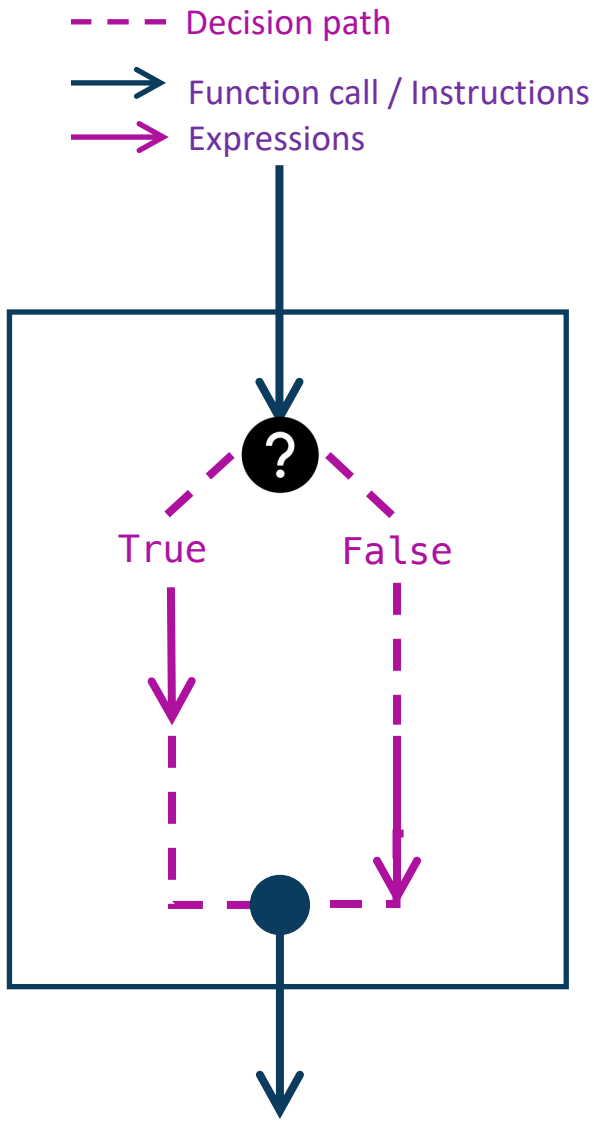
$a^2 + b^2 = c^2$

```
1  def check_pythagorean(a, b, c):
2      if a*a + b*b == c*c:
3          print("The 'if' condition is met")
4          return 'yes'
5      else:
6          print("The 'if' condition is not met")
7          return 'no'
8
9  result = check_pythagorean(3, 4, 5)
10 print(result)
11 result = check_pythagorean(5, 12, 13)
12 print(result)
13 result = check_pythagorean(6, 8, 20)
14 print(result)
```

```
def func_name(parameters)
    if <condition>:
            <expression>
            <expression>
            return value
    else:
            <expression>
            <expression>
            return value
```

```
func_name(arguments)
<instructions>

func_name(arguments)
<instructions>
. . .
```

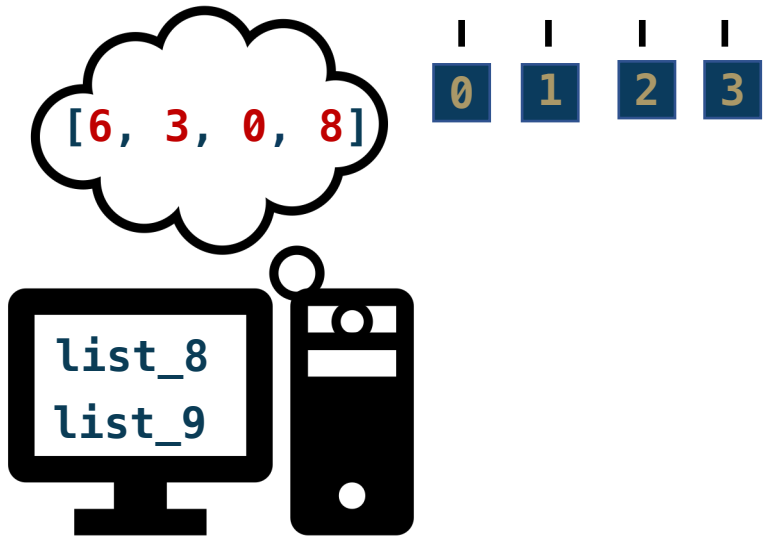
Demo



Lists are mutable!

Sources:
Tuples, Lists, Aliasing and Mutability By Ana Bell: https://youtu.be/RvRKT-jXvko
Functions: https://www2.seas.gwu.edu/~cs4all/1012/unit1/module1.2.html

```
>>> list_8 = [6, 3, 0, 8]
```



Key thing to remember is that variables of lists may be affected by the changes or mutations

```
1  def swap_list_first(A, B):
2      temp = A[0]
3      A[0] = B[0]
4      B[0] = temp
5
6  X = [1, 3, 5, 7]
7  Y = [2, 4, 6]
8  swap_list_first(X, Y)
9  print(X, Y)
```

A = X B = Y

X[0] 1 A[0] 1 temp 1

Y[0] 2 B[0] 2 A[0] 2

temp 1 B[0] 1

X: X[0] X[1] X[2] X[3] = 1 3 5 7

Y: Y[0] Y[1] Y[2] = 2 4 6

X: A[0] A[1] A[2] A[3] = 2 3 5 7 Y: B[0] B[1] B[2] = 1 4 6

```
>>> list_9 = list_8
>>> list_8[2] = 7
>>> list_8
[6, 3, 7, 8]
```

```
>>> list_9
[6, 3, 7, 8]
```

Demo

School of Engineering & Applied Science
THE GEORGE WASHINGTON UNIVERSITY

GW

Calling functions from functions

Data analysis

This part of code calculates Mean or Average of a list containing numbers.

```python
import math

def compute_mean(A):
    # Insert your code here:

def compute_std(A):
    mean = compute_mean(A)
    total = 0
    for k in A:
        total += (k-mean)**2
    std = math.sqrt(total / len(A))
    return std

data = [-2.3, -1.22, 1.6, -10.5, 1.4, 2.5, -3.32, 11.03, 2, 2, -1.4]
print('mean =', compute_mean(data))
print('standard deviation =', compute_std(data))
```

The mean is the average of all numbers and is sometimes called the arithmetic mean.

$$mean = \frac{sum\ of\ elements\ in\ a\ list}{number\ of\ elements\ in\ the\ list} = \frac{sum(A)}{len(A)}$$

This part of code calculates the standard deviation of the list containing numbers.

- It requires the mean of the list as input

The standard deviation is a measure of the amount of variation or dispersion of a set of values.

$$std = \sqrt{\frac{sum\ of (A[k] - mean)^2}{number\ of\ elements\ in\ the\ list}}$$

Demo

Functions … calling itself!

The **factorial** of a non-negative integer n,

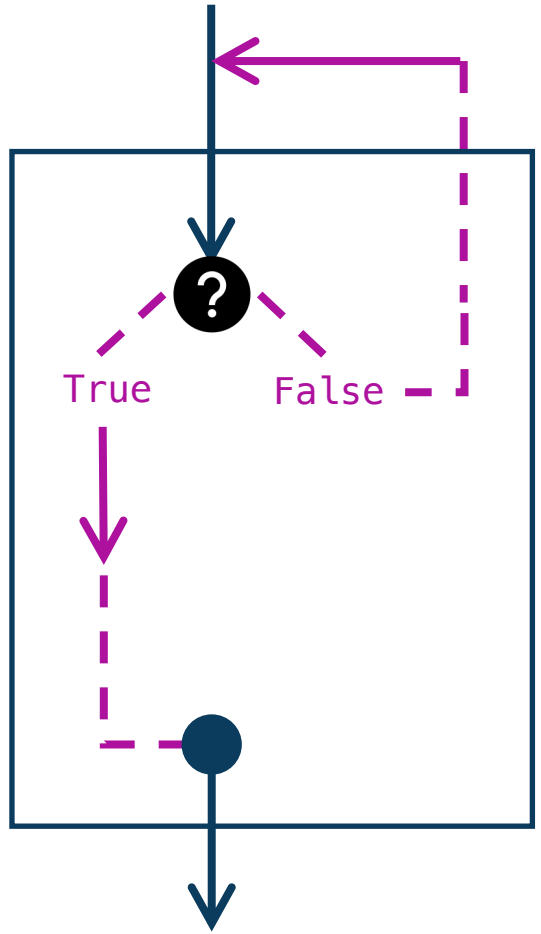
$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 3 \times 2 \times 1$$

- is denoted by $n!$ *and*
- is the product of all positive integers less than or equal to n.

$$3! = 3 \times 2 \times 1$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

```
def factorial(n):
    # print(n)
    if n == 1:
        return 1
    else:
        m = factorial(n-1)
        return n * m

print(factorial(3))
print(factorial(5))
```



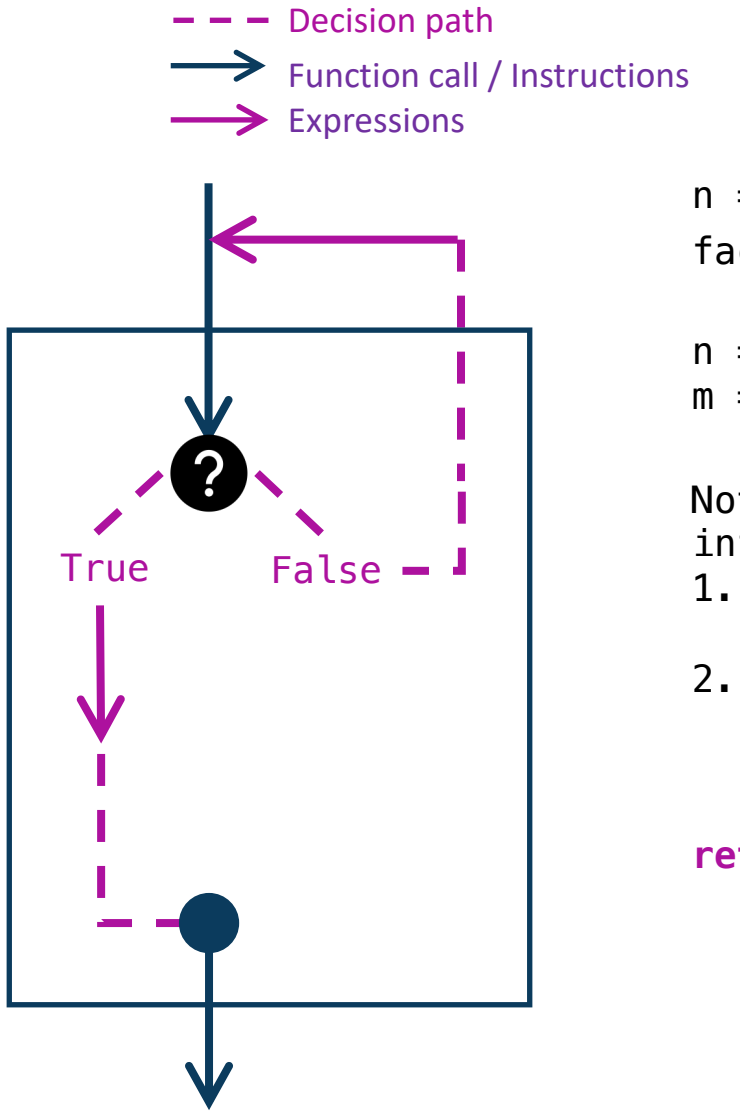
Key ideas

- Recursion is the process of defining something in terms of itself.
- A function that calls itself is said to be recursive.
- When a function calls itself, that's called a recursion step.

Demo

Functions … calling itself!

```
def factorial(n):
    # print(n)
    if n == 1:
        return 1
    else:
        m = factorial(n-1)
        return n * m

print(factorial(3))
print(factorial(5))
```

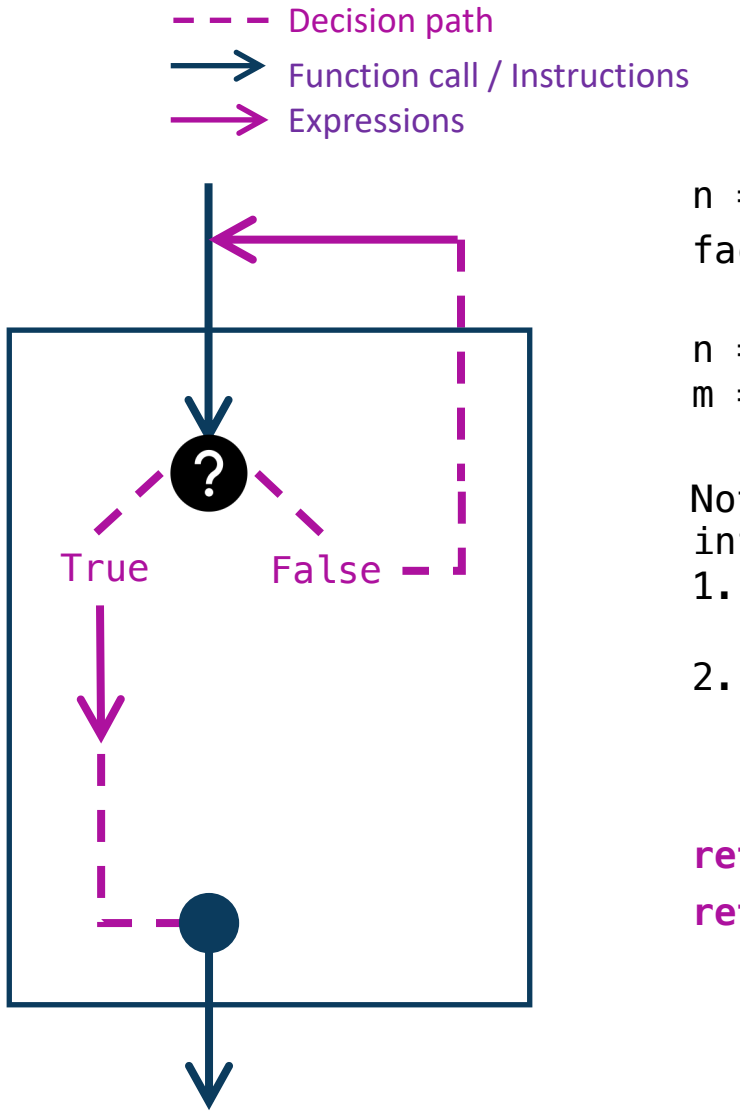



```
n = 3
factorial(3)

n == 1 → False
m = factorial(2)

Note: The program collects
information that
1. It needs to compute
   factorial(2)
2. It needs to circle back
   and call the factorial()
   again.

return 3 * factorial(2)
```

Demo

Functions … calling itself!

```
def factorial(n):
    # print(n)
    if n == 1:
        return 1
    else:
        m = factorial(n-1)
        return n * m

print(factorial(3))
print(factorial(5))
```

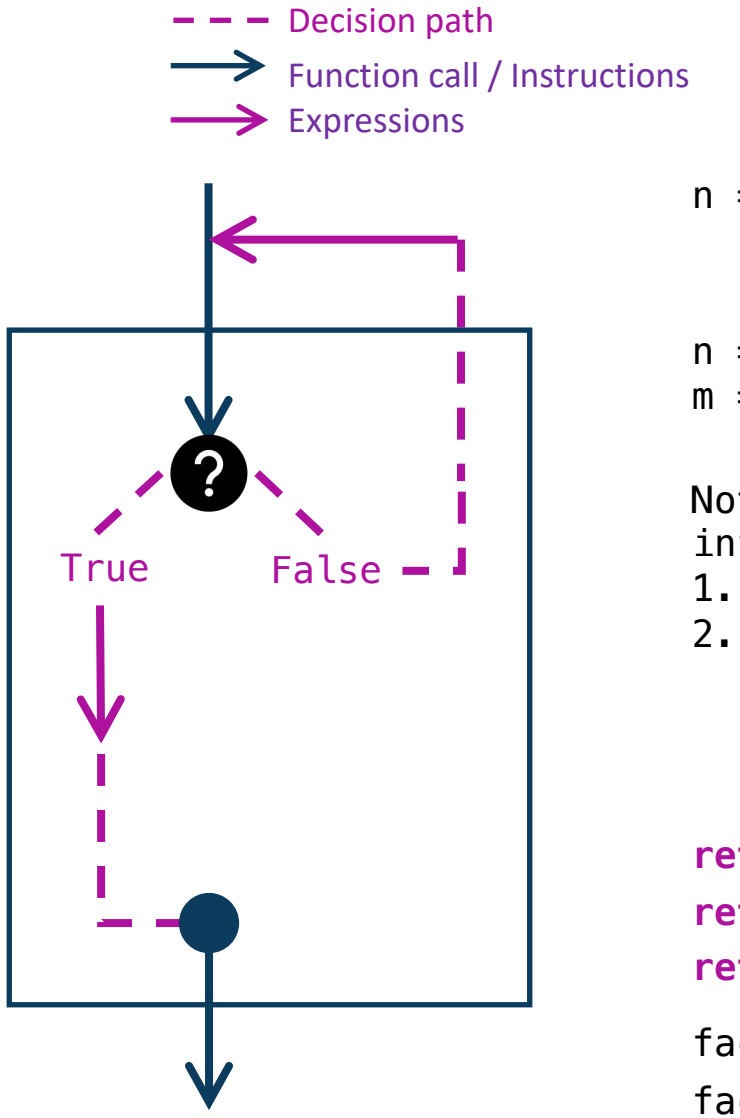



```
n = 2
factorial(2)

n == 1 → False
m = factorial(1)

Note: The program collects
information that
1. It needs to compute
   factorial(1)
2. It needs to circle back
   and call the factorial()
   again.

return 3 * factorial(2)
return 2 * factorial(1)
```

Demo

Functions … calling itself!

```
def factorial(n):
    # print(n)
    if n == 1:
        return 1
    else:
        m = factorial(n-1)
        return n * m

print(factorial(3))
print(factorial(5))
```




```
n = 1

n == 1 → True
m = factorial(1)

Note: The program collects
information that
1. factorial(1) is 1
2. It needs to execute all
   expressions stored in
   memory

return 3 * factorial(2)
return 2 * factorial(1)
return 1

factorial(3) = 3 * 2 = 6
factorial(2) = 2 * 1 = 2
factorial(1) = 1
```

Demo

Key take-home ideas:

- Functions are defined before they are called
- Functions can be called any number of times once defined
- Single or multiple arguments can be passed into a function
- Any variables assigned inside the function will not be accessible outside the function
- Functions can return value or return nothing
- A Function can be called within itself

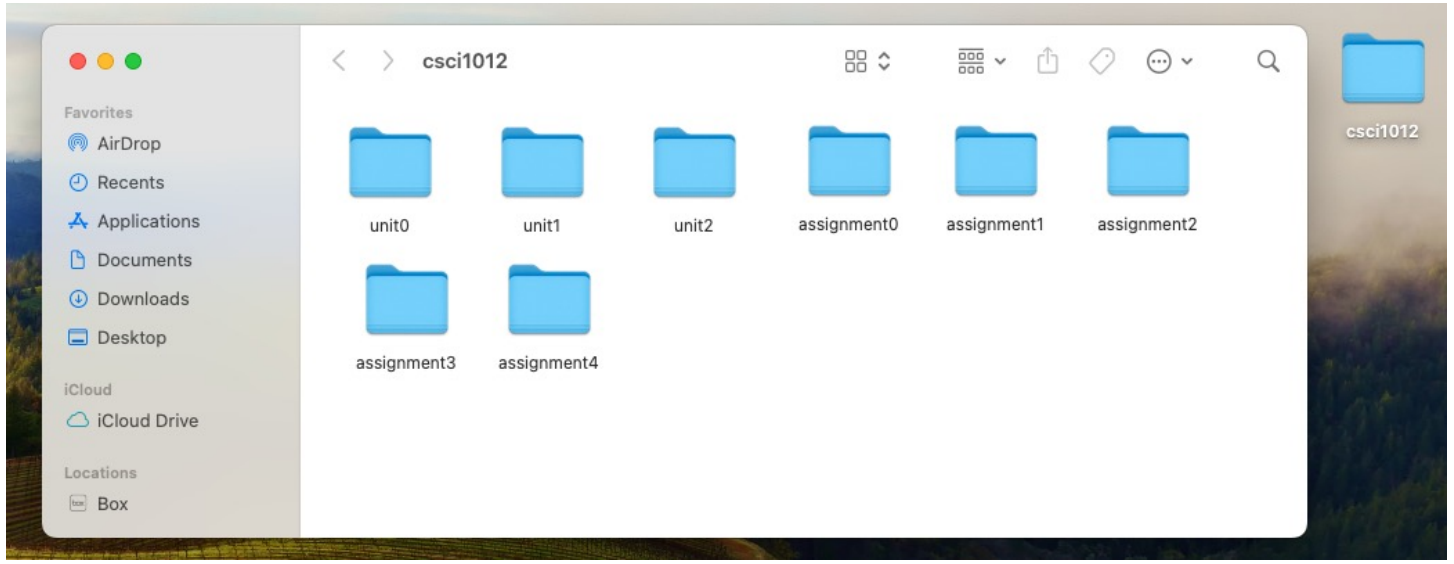
File-folder-structure

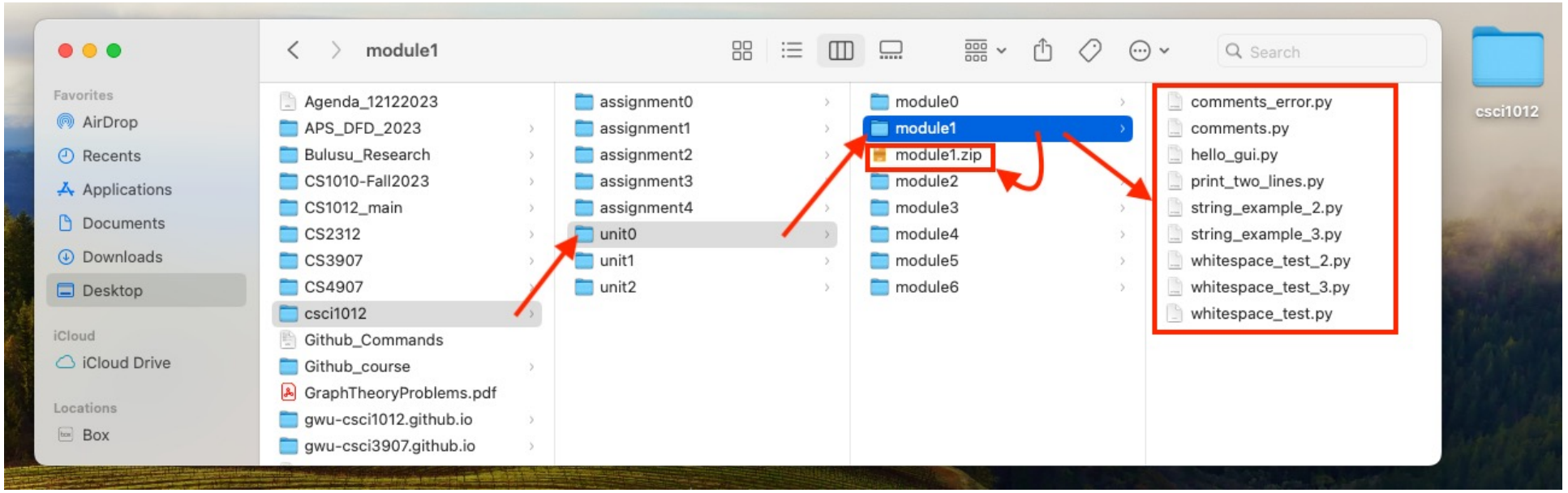
`module0.zip` (correct)

`Module0.zip` (wrong: starts with uppercase)

`module 0.zip` (wrong: space before 0)

`module0.docx` (wrong: not a zip).





See you all in the Wednesday and Friday Labs!