

Computational Thinking

## Discrete Mathematics

Number Theory

Topic 01 : Computational Thinking

Logic

### Lecture 01 : Fundamentals of Computation

Dr Kieran Murphy 

Computing and Mathematics, SETU (Waterford).  
(kieran.murphy@setu.ie)

Graphs and  
Networks

Autumn Semester, 2025/26

Collections

#### Outline

- Using PyTutor with Colab
- Python Fundamentals
- Storing data and data types, Making decisions, Looping, Functions

Enumeration

Relations & Functions

# Outline

---

1. Using PyTutor with Colab	2
2. Python Fundamentals	8
2.1. History of Python	9
2.2. First Look at Python Code	10
2.3. Data and Data Types	11
2.4. Collections	13
2.5. Looping	16
2.6. Making Decisions	18
2.7. Functions	20

# Using PyTutor with Colab

I

Before we start covering Python we want to show you PyTutor in action. The following slides shows screenshots of the process but you should verify the steps yourself on your phone/tablet.

## Step 1 — Click/Scan on QR Code

The following code outputs powers of 2, don't worry about the actual code, just make sure that you can open and use PyTutor ...

```
1 powers = [0,1,2,3,4,5,6]
2 for p in powers:
3     print(p, 2**p)
```

```
0 1
1 2
2 4
3 8
4 16
5 32
6 64
```

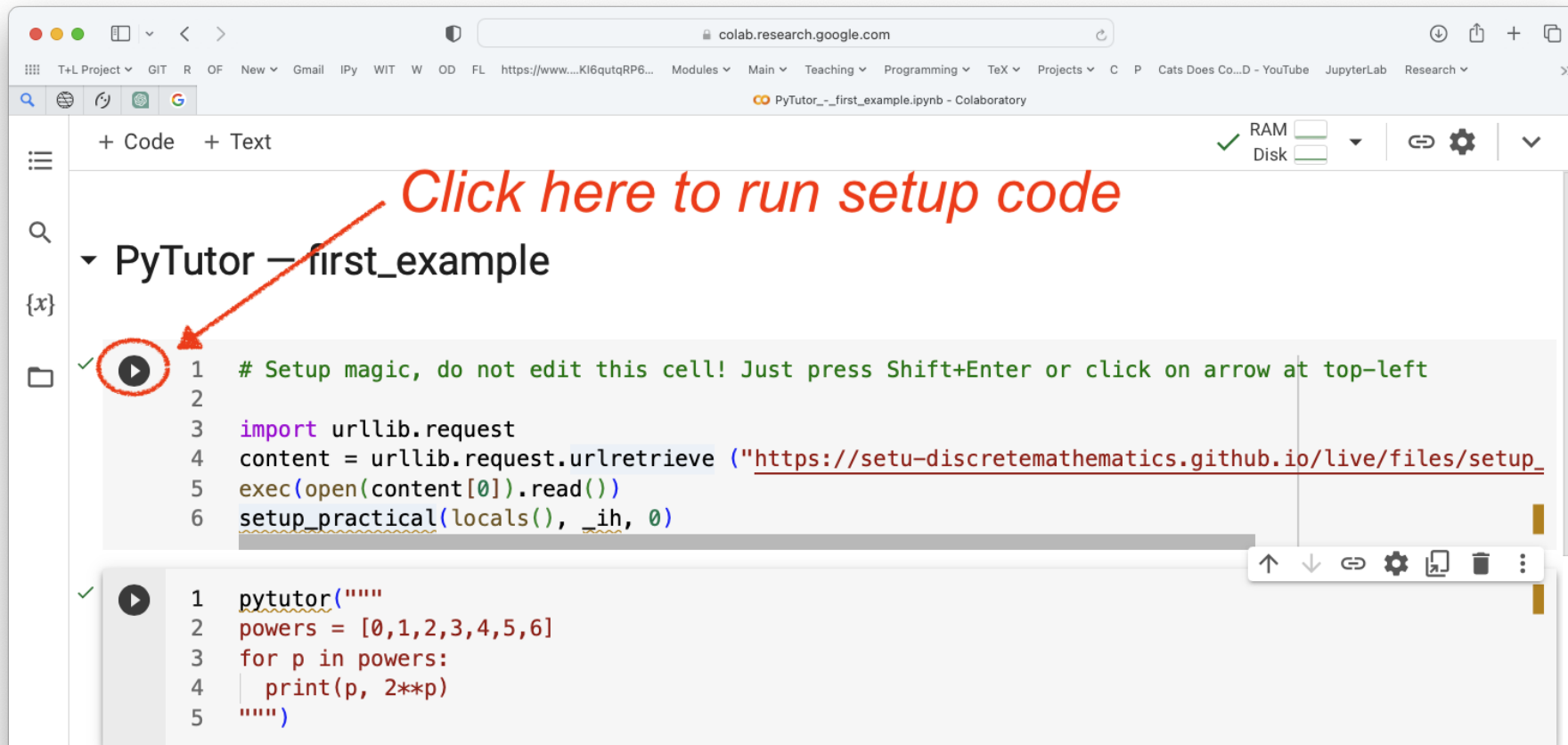


# Using PyTutor with Colab

This should open in Colab the following notebook.

Unlike our practical notebooks, don't bother clicking on **File** → **Save a copy in Drive**.

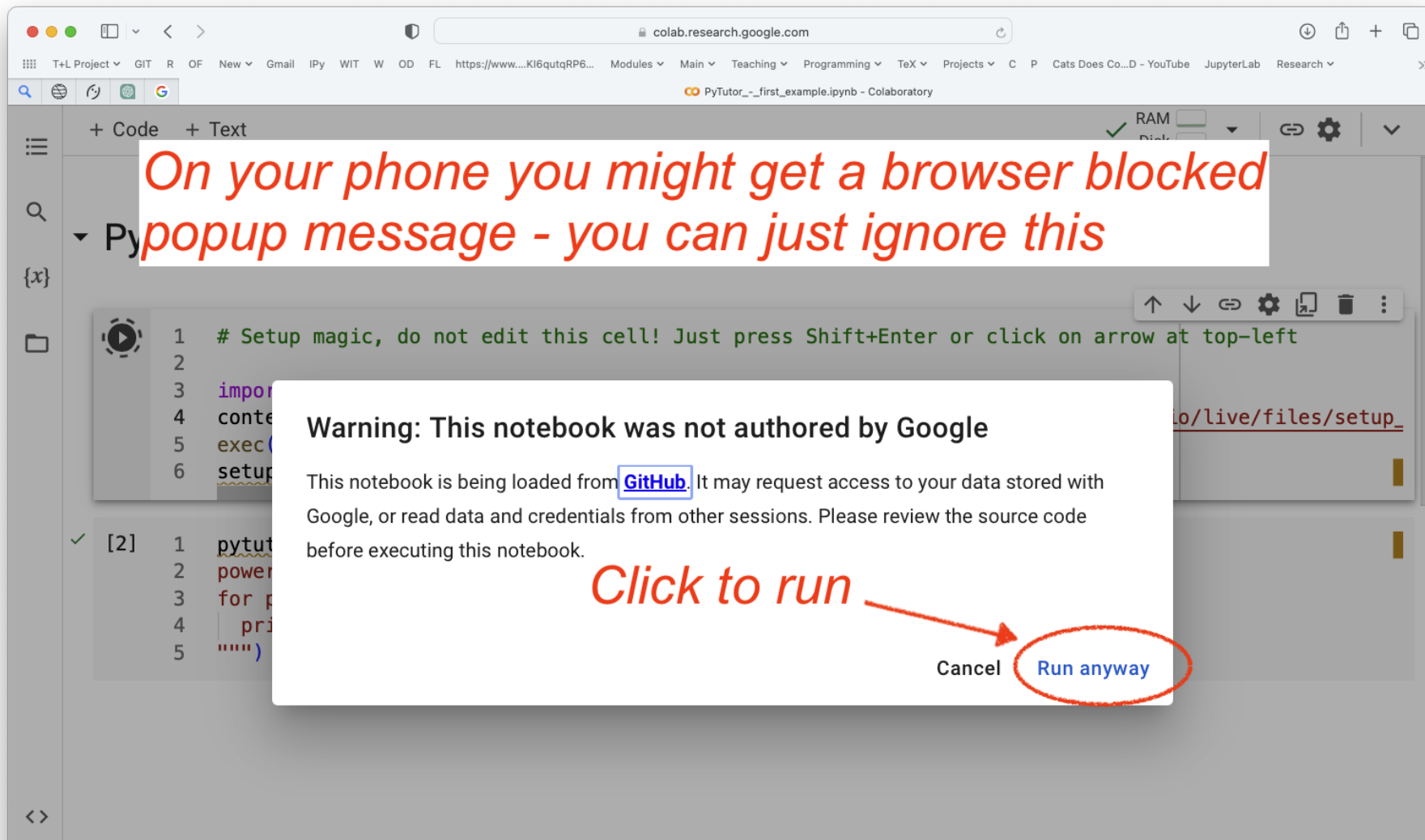
**Step 2 — Execute the first cell to setup notebook.**



# Using PyTutor with Colab

On executing the first cell you will get the following message. Click on **Run anyway**.

Step 3 — Click on Run anyway



# Using PyTutor with Colab

After executing the first cell you will get the usual "Python practical setup tools version 23.2".

Step 4 — Click on second cell to run code in PyTutor

colab.research.google.com

PyTutor\_-\_first\_example.ipynb - Colaboratory

+ Code + Text Cannot save changes

RAM ☐ Disk ☐

PyTutor — first\_example

Click here to start pytutor

```
1 # Setup magic, do not edit this cell! Just press Shift+Enter or click on arrow at top-left
2
3 import urllib.request
4 content = urllib.request.urlretrieve ("https://setu-discretemathematics.github.io/live/files/setup_
5 exec(open(content[0]).read())
6 setup_practical(locals(), _ih, 0)
```

Python practical setup tools version 23.2. See [https://setu-discretemathematics.github.io/live/00-Module\\_Introduction/33-Python\\_Practicals](https://setu-discretemathematics.github.io/live/00-Module_Introduction/33-Python_Practicals)

[2] 1 pytutor("""
2 powers = [0,1,2,3,4,5,6]
3 for p in powers:
4 | print(p, 2\*\*p)
5 """)

# Using PyTutor with Colab

You can now use PyTutor, to step back/forward through the code and see the current data values and resulting output.

The screenshot displays the PyTutor interface integrated into a Google Colab notebook. The interface is divided into several sections:

- 1. Code:** A code editor showing Python 3.6 code. The code is:
 

```
1 powers = [0,1,2,3,4,5,6]
2 for p in powers:
3     print(p, 2**p)
```

 The code is annotated with a green arrow pointing to line 2 (indicating the line just executed) and a red arrow pointing to line 3 (indicating the next line to execute).
- 2. Step controls:** A control panel at the bottom left of the code editor. It includes a play button, a slider for stepping through the code, and buttons for "< Prev" and "Next >". The text "Step 9 of 16" is displayed below the slider.
- 3. Data values:** A section on the right side of the interface showing the current state of the program's memory. It includes a "Frames" pane showing the "Global frame" with variables "powers" and "p" (value 3). An "Objects" pane shows a "list" object containing the values [0, 1, 2, 3, 4, 5, 6].
- 4. Output:** A text area on the right side of the interface showing the output of the code execution. The output is:
 

```
0 1
1 2
2 4
```

Red handwritten annotations are present on the image, circling the code editor, the step controls, the data values section, and the output area, and labeling them with the numbers 1, 2, 3, and 4 respectively.

# Outline

---

1. Using PyTutor with Colab	2
2. Python Fundamentals	8
2.1. History of Python	9
2.2. First Look at Python Code	10
2.3. Data and Data Types	11
2.4. Collections	13
2.5. Looping	16
2.6. Making Decisions	18
2.7. Functions	20



## Brief History of Python

- Invented in the Netherlands, early 90s by Guido van Rossum.

*“Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another’s code; too little and expressiveness is endangered.”*

– Guido

- Named after Monty Python.
- Scalable, object oriented and functional from the beginning.
- Python 3.0 was released in 2008, to rectify certain flaws in Python 2.\*.
- Most popular language for machine learning and data mining.

### Python’s Benevolent Dictator For Life



# First Look at Python Code

To get an idea of Python, we will take a small piece of code\*

```
1  # Solution to Euler problem 2
2
3  # Calculate the sum of the even-values in the Fibonacci sequence
4  #    1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
5  # value that do not exceed four million,
6
7  last = 1
8  current = 2
9
10 answer = 0
11 while current <= 4_000_000:
12     if current % 2 == 0:
13         answer += current
14     last, current = current, last + current
15
16 print(answer)
```

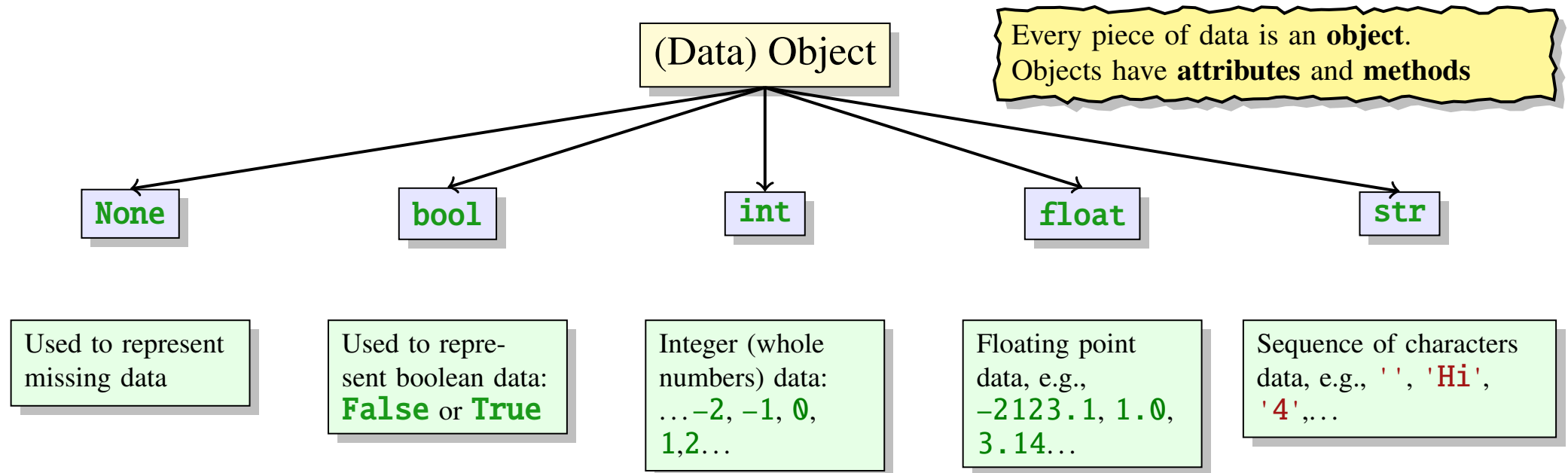


\*This is a solution to the [Euler Problem 2](https://projecteuler.net/problem=2), at the programming competition site, [projecteuler.net](https://projecteuler.net).

# Data and Data Types

I

Python has 5 main primitive data types:



- An **Object** stores data in its **attributes**, and **methods** are used to change an object.
- In Python, the type of the data is automatically determined (unlike Processing).
- The type determines what you are allowed to do to a piece of data.
- Function **type** will return the type of a piece of data.

# Data and Data Types

## II

```
3 ●
1 w = None
2
3 x = 4
4 y = '4'
5 z = 4.0
6
7 print(type(w), type(x), type(y), type(z))
8
9 x = x * 10
10 y = y * 10
11 z = z * 10
12
13 x = x * 1_000_000_000
14 z = z * 1_000_000_000
15
16 x = x / 1_000_000_000
17 z = z / 1_000_000_000
18
19 print(type(w), type(x), type(y), type(z))
```



## Collections: **set**, **list**

We will cover collections in more detail later, but for now we have:

### Sets

- A **set** is collection of **distinct**, **unordered** values.
  - **distinct** means a set cannot hold the same piece of data more than once.
  - **unordered** means we cannot sort the elements of a set or ask "what element is first?" etc.
  - We can manipulate sets using union **|**, intersection **&**, and set minus **-** operations.

### Lists

- A **list** is collection of **ordered** values.
  - **ordered** means the values appear in a sequence (i.e., have position). So we can talk about which value appears before (or after) another value. (**ordered**  $\neq$  **sorted**)
  - Data values do not have to be distinct.
  - The position of a data value in a list is called its **index**. Since Python is a **zero-based language**, the position starts at 0.
  - Lists are a BIG DEAL in python and we have many operations to manipulate them (more later).

# Collections: set

```
4 ●
1 z = set() # creating a empty set
2
3 # defining sets by stating values
4 a = {1, 3, 1, 2, 1, 5, 4}
5 b = {1, 'a', 3}
6
7 print(len(a)) # size of set
8
9 c = a & b # intersection
10 print(c)
11
12 c = a | b # union
13 print(c)
14
15 c = a - b # set difference
16 print(c)
17
18 c = b - a # set difference
19 print(c)
```



# Collections: **list**

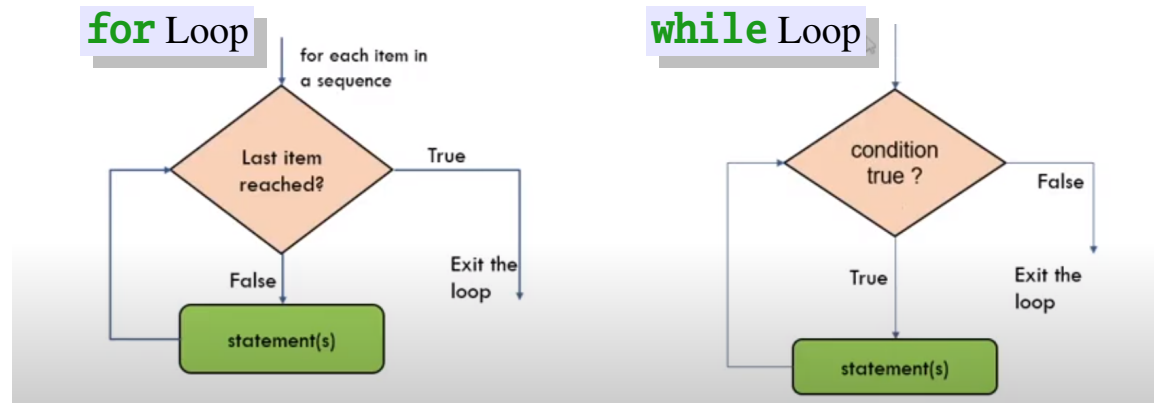
```
5 1 z = [] # creating a empty list
2
3 # defining lists by stating values
4 a = [1,3,1,2,1,5,4]
5 b = [1,3]
6
7 print(len(a)) # size of list
8
9 c = a + b # appending lists
10 print(c)
11
12 value = c[2] # list indexing ZERO-BASED
13 print(value)
14
15 d = c[2:5] # slicing SEMI-OPEN notation
16 print(d)
17
18 value = c[-4] # negative indexing
19 print(value)
```



# Looping: **for**, **while**

**for** — Looping when you know how many times you want to repeat

- Python **for** loop is actually a **for-each** loop.
  - In a for-each loop you loop over values in a collection. This is considered to be less error prone than standard for loops. (They are more likely to have **off-by-one errors**.)
  - Python has function **range** to efficiently build collections to be used in **for** loops.



**while** — Looping when you don't know how many times you want to repeat

- In a **while** loop, since we don't know how many times to loop, we have to define a **stopping condition**.
  - A **while** loop will keep repeating a block of code **while** the **condition** calculates to a **True** value.



# Looping: **for** Loop

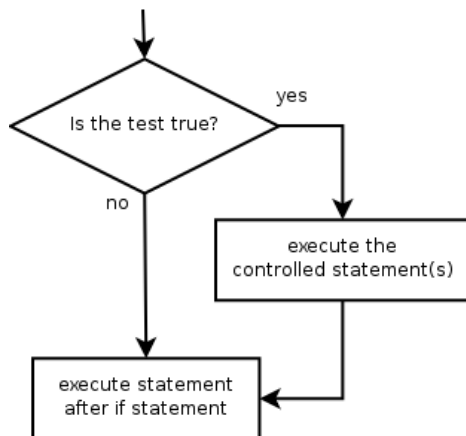
```
6 ●
1  # for loops runs over a collection
2
3  print('Looping over a list')
4  for letter in ['a', 'e', 'i', 'o', 'u']:
5      print(letter, 'is a vowel')
6
7  # BUT be careful if the collection is a set
8  # since a set does not have order
9  print('Looping over a set')
10 for letter in {'a', 'e', 'i', 'o', 'u'}:
11     print(letter, 'is a vowel')
12
13 # Function range is useful in creating collections
14 # NOTE Python uses SEMI-OPEN intervals !!!
15 print('Use range to build collections')
16 for x in range(5):
17     print(x)
```



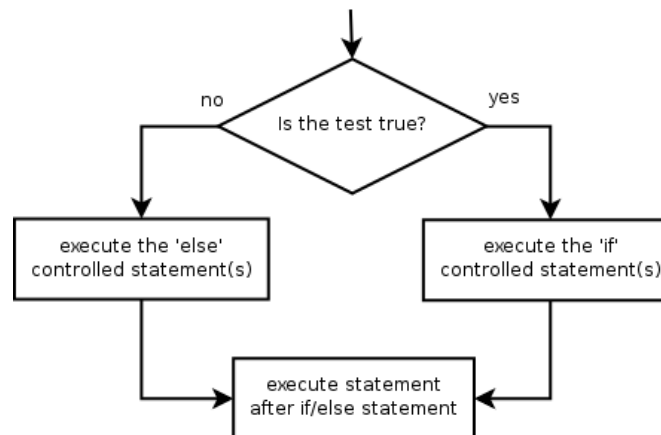
# Making Decisions: **if**, **elif**, **else**

The **if** statement controls which blocks of code to execute based on given conditions. It has three variations:

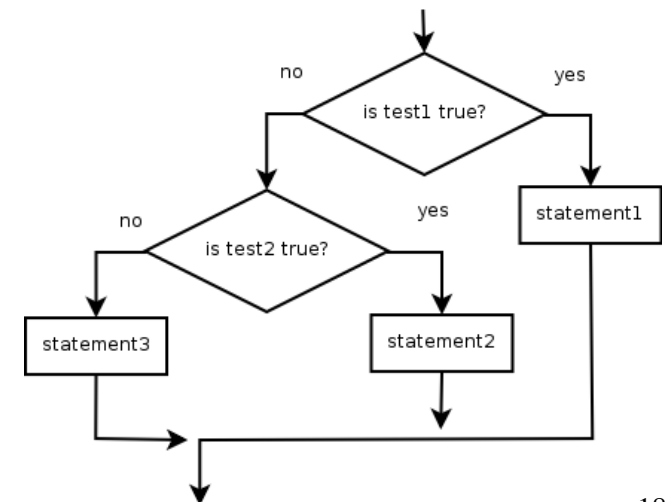
```
if test:
    statements
```



```
if test:
    if_statements
else:
    else_statements
```



```
if test_1:
    statements_1
elif test_2:
    statements_2
else:
    statements_3
```



## Making Decisions: **if**, **elif**, **else**

```
7 # In the drinking game of fuzz-buzz players count in turn  
1 # but replace multiples of 3 with 'fuzz',  
2 # multiples of 5 with 'buzz',  
3 # and multiples of 15 with 'fuzz buzz'  
4  
5  
6 for k in range(1,21):  
7     if k%15==0: # is k a multiple of 15?  
8         print("fuzz buzz")  
9     elif k%3==0: # is k a multiple of 3?  
10        print("fuzz")  
11    elif k%5==0: # is k a multiple of 5?  
12        print("buzz")  
13    else:  
14        print(k)
```



## Functions: `def`, `return`

- In Python a function is a block of code defined with a name — this allows us to reuse code and improve code quality.
- A function is a block of code that only runs when it is called.
- You pass data, known as **parameters**, into the function. And pass data back using `return`.

The diagram illustrates a Python function definition and its usage. The function definition is as follows:

```
8 def add(num1, num2):  
1  
2  
3     print("Number 1", num1)  
4     print("Number 2", num2)  
5     result = num1 + num2  
6  
7     return result
```

Annotations for the function definition:

- Function name:** Points to `add`.
- parameters:** Points to `num1, num2`.
- function body:** A bracket indicates the block of code from `print("Number 1", num1)` to `return result`.
- Return value:** Points to `result` in the `return` statement.

The function is called in the following code:

```
9 ans = add(5, 7)  
10 print("Function returned", ans)
```

Annotations for the function call:

- Function call:** Points to `add(5, 7)`.

