

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, 2023

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
```

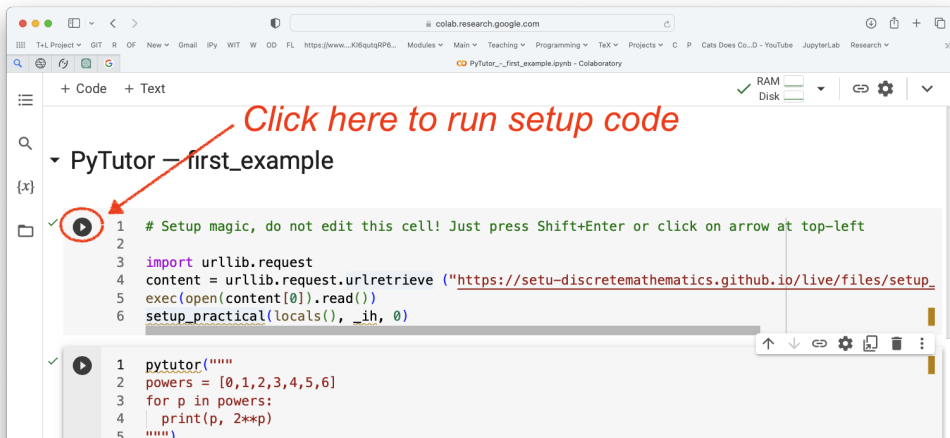


Using PyTutor with Colab

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

The screenshot shows a Google Colab notebook in a web browser. A warning dialog box is displayed in the center, with the text: "Warning: This notebook was not authored by Google. This notebook is being loaded from [GitHub](#). It may request access to your data stored with Google, or read data and credentials from other sessions. Please review the source code before executing this notebook." Below the dialog box, the text "Click to run" is written in red, with a red arrow pointing to the "Run anyway" button, which is circled in red. The background shows the notebook interface with a code cell containing Python code for setting up PyTutor.

On your phone you might get a browser blocked popup message - you can just ignore this

```
1 # Setup magic, do not edit this cell! Just press Shift+Enter or click on arrow at top-left
2
3 import sys
4 content = sys.stdin.read()
5 exec(content)
6 setup_py_tutor()
```

[2] 1 pytu
2 power
3 for p
4 pr
5)

Warning: This notebook was not authored by Google

This notebook is being loaded from [GitHub](#). It may request access to your data stored with Google, or read data and credentials from other sessions. Please review the source code before executing this notebook.

Click to run

Cancel **Run anyway**

Using PyTutor with Colab

IV

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

The screenshot shows a Google Colab notebook interface. The browser address bar is `colab.research.google.com`. The notebook has a tab titled `PyTutor_-_first_example.ipynb - Colaboratory`. The interface includes a top bar with icons for RAM and Disk usage, and a left sidebar with icons for file explorer, search, and code execution. The notebook content is divided into two cells under the heading `PyTutor — first_example`.

The first cell contains the following code:

```
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)
```

Below the first cell, the output is displayed: `Python practical setup tools version 23.2. See https://setu-discretemathematics.github.io/live/00-Module_Introduction/33-Python_Practicals`.

The second cell is selected and contains the following code:

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

A red arrow points from the text *Click here to start pytutor* to the first cell's run button (a black triangle icon).

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.

colab.research.google.com

PyTutor_-_first_example.ipynb - Colaboratory

Cannot save changes

RAM ☐ Disk ☐

1. Code

```
Python 3.6
1 powers = [0,1,2,3,4,5,6]
2 for p in powers:
3     print(p, 2**p)
```

Edit this code

→ line that just executed
→ next line to execute

2. Step controls

Step 9 of 16

4. Output

Print output (drag lower right corner to resize)

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

3. Data values

Frames

Global frame

powers

p 3

Objects

list

0	1	2	3	4	5	6
0	1	2	3	4	5	6

Set AI Help

Move and hide objects

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.

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
3 # Calculate the sum of the e
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)
```

The character # indicates an end of line comment.
In each line everything after the # is ignored by the computer



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

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
4 # 1, 2, 3, 5, 8, 13, 21,
5 # value that do not exceed
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)
```

Use `=` to store data.

On the right of `=`, we have the data value(s)

On the left of `=`, we have the **identifier** name(s)

Unlike other languages (e.g., Processing) we don't need to state the data type. (More on this later.)



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

First Look at Python Code

To get an idea of Python,

```
1 # Solution to Euler problem 2
2
3 # Calculate the sum of the
4 # 1, 2, 3, 5, 8, 13, 21, ...
5 # value that do not exceed
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)
```

A core feature of Python is **indenting**

indent is the spacing at **start** of Python lines of code. It is used to specify blocks of code, for functions, loops or decisions.

Here we have a **while** loop block with lines 12–14. Inside that, we have an **if** decision block with line 13.

Note the **:** at end of line **before** code block

Other languages (e.g., Processing) use brackets **{** and **}** to specify blocks. Python doesn't and this results in cleaner code.



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

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
4  #   1, 2, 3, 5, 8, 13, 21,
5  # value that do not exceed
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)
```

Python has lots of little features that make coding nicer.

For example:

- We can use underscore `_` to represent thousand separator in numbers (line 11).
- We can assign multiple values at the same time (line 14).

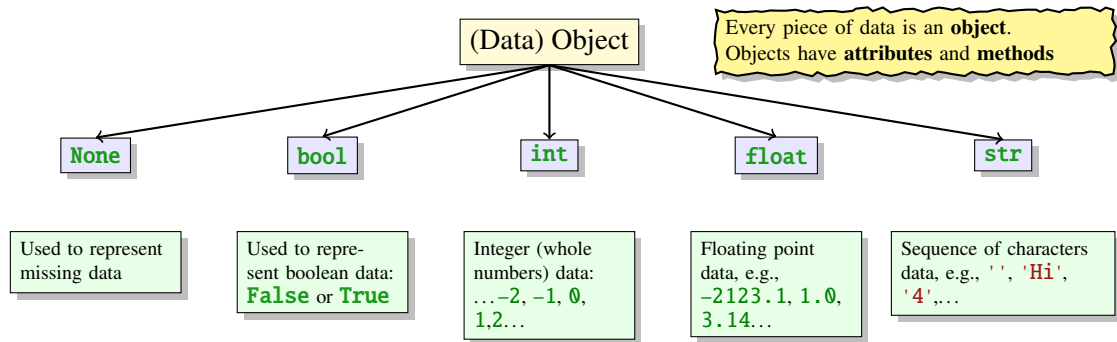


*This is a solution to the [Euler Problem 2](https://projecteuler.net/problem=2), at the programming competition site, projecteuler.net.

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

```
7 ●
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))
```



Data and Data Types

```
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))
```

Python infers the type from the data

- **str** data is indicated by single or double quotes.

- **float** data has a decimal point.

or from the result of an operation on data.

- **int** divided by an **int** becomes a **float**. Why?



Data and Data Types

Operations (here multiplication using `*`) can do different things based on the type.



```
9 ●
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))
```

Data and Data Types

Using function **type** on an object is a common feature of Python programming.

```
10 ●
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 of 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**

```
11 ●
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: **set**

Have special notation for creating empty sets (We can't use {}, more on this later).

```
12 ● 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: **set**

```
13 ● 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)
```

Use { and } to define a **set**.

Repeated value are ignored.

Order does not matter.

Collections in Python can store a mixture of types (this is really useful!)



Collections: **set**

Python **set** supports the standard operations you know (and love) from Mathematics.



```
14 ● 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**

```
15 ●
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)
```



Collections: **list**

Use square brackets, `[]`, to create empty lists.

```
16 ● 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)
```



Collections: **list**

```
17 ●
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
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)
```

Use `[` and `]` to define a **list**.

Repeated value are allowed.

Order does matter (in fact is it usually important).

Collections in Python can store a mixture of types
(this is really useful!)



Collections: **list**

```
18 ●
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
8
9  c = a + b # appending list
10 print(c)
11
12 value = c[2] # list indexing
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)
```

Use `[` and `]` to define a **list**.

Because data values in a **list** have position (**index**) we can access particular value(s) using

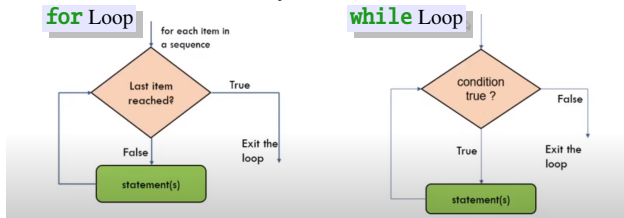
- **indexing**: to access a single data value
- **slicing**: building a new **list** by taking some values from a **list**



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

```
19 ●
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)
```



Looping: `for` Loop

20 ●

```
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)
```

`for` loop can run over any collection, even a `set`.
But since a `set` does not have order, the result may be surprising.



Looping: `for` Loop

21 ●

```
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 range
8 # since a set does not have an end value
9 print('Looping over a sequence')
10 for letter in 'aeiou':
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)
```

Function **range** is typically used to generate collections of integers.

Python uses **semi-open** intervals — this means the starting value is included but the end value is excluded.

range(END)

0, 1, 2, 3, ..., <END

range(START, END)

START, START+1, START+2, ..., <END

range(START, END, STEP)

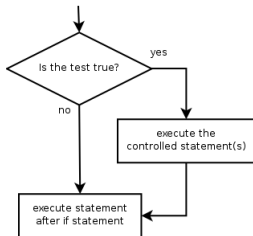
START, START+STEP, START+2*STEP, ..., <END



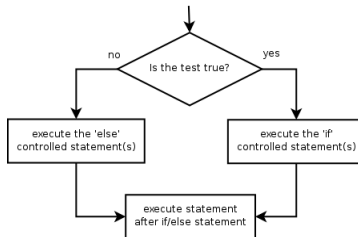
Making Decisions: `if`, `elif`, `else`

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

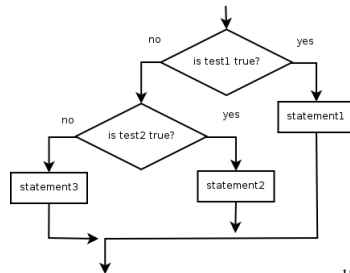
```
if test:  
    statements
```



```
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**

22 ●

```
1  # In the drinking game of fuzz-buzz players count in turn
2  # but replace multiples of 3 with 'fuzz',
3  # multiples of 5 with 'buzz',
4  # and multiples of 15 with 'fuzz buzz'
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)
```



Making Decisions: `if`, `elif`, `else`

```
23 ●
1  # In the drinking game of fuzz buzz
2  # but replace multiples of 3 with 'fuzz'
3  # multiples of 5 with 'buzz'
4  # and multiples of 15 with 'fuzz buzz'
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)
```

`range(1,21)` is a collection of `int` starting at 1 (inclusive) up to end at 21 (exclusive)

- We use `==` to test for equality.
- `%` is the modulus operator. It returns the remainder on division.

Why did we test `k` was a multiple of 15 first?



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`.

24 ●

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

Diagram labels:

- Function name: points to `add`
- parameters: points to `num1, num2`
- function body: points to the block of code between `def` and `return`
- Return value: points to `result`
- Function call: points to `add(5, 7)`

