Discrete Mathematics

Topic 01 : Computational Thinking

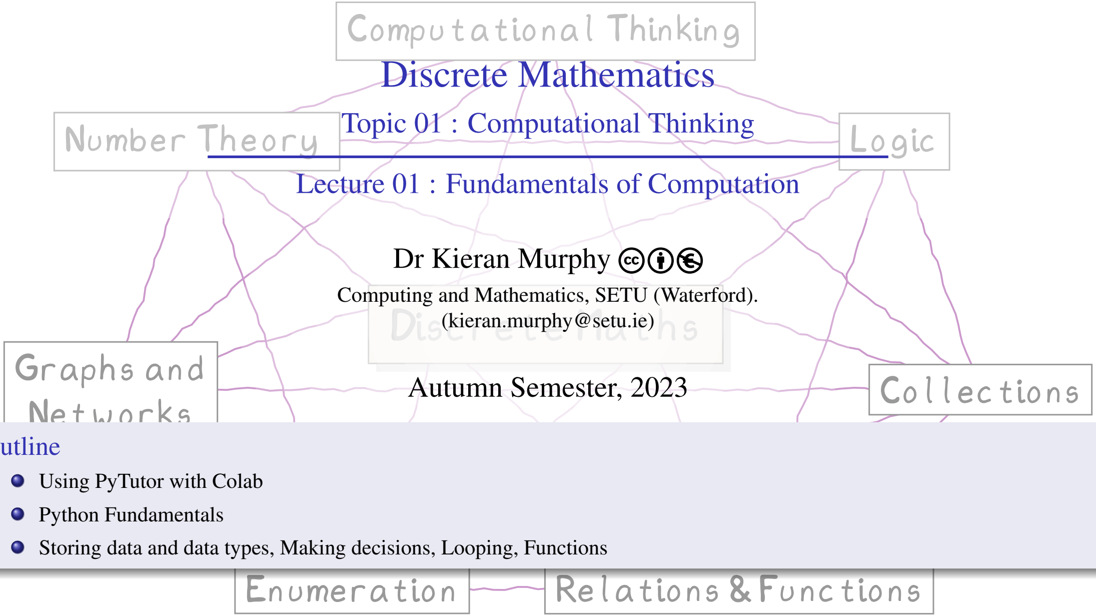
Lecture 01 : Fundamentals of Computation

Dr Kieran Murphy

Computing and Mathematics, SETU (Waterford).

(kieran.murphy@setu.ie)

Autumn Semester, 2023



Outline

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

Outline

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 . . .

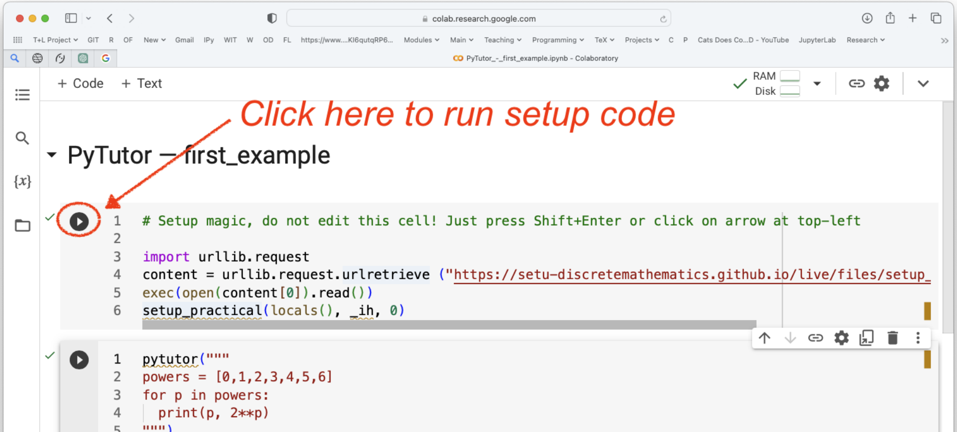
```python
powers = [0,1,2,3,4,5,6]
for p in powers:
  print(p, 2**p)
```

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

Using PyTutor with Colab II

This should open in Colab the following notebook.
Unlike our practical notebooks, don't bother clicking on File → Save a copy in Drive.

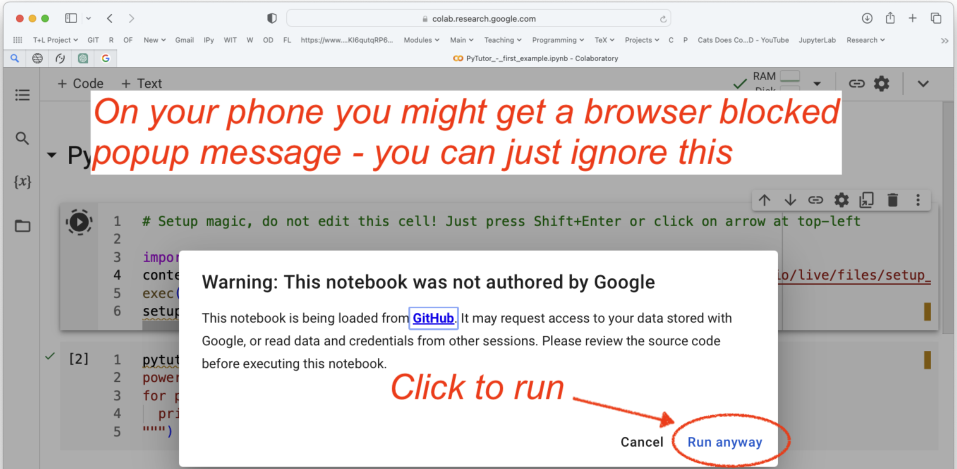
Step 2 — Execute the first cell to setup notebook.



Using PyTutor with Colab III

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

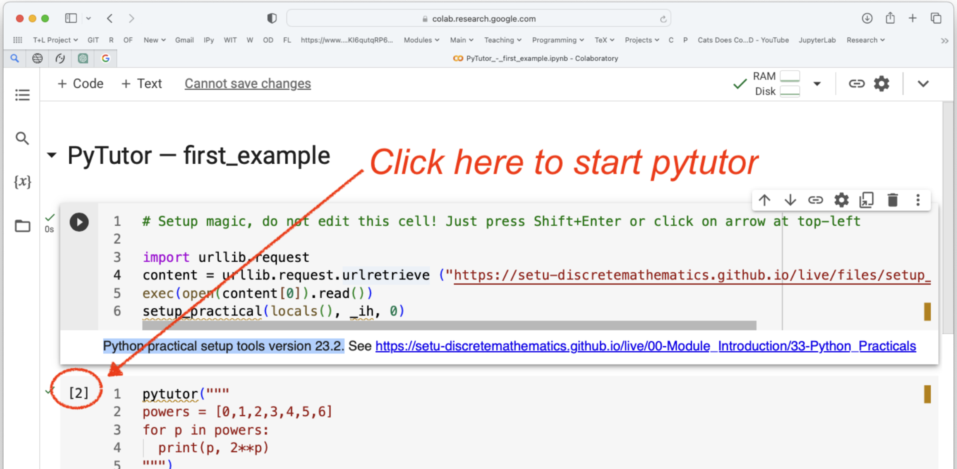
Step 3 — Click on Run anyway



Using PyTutor with Colab IV

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

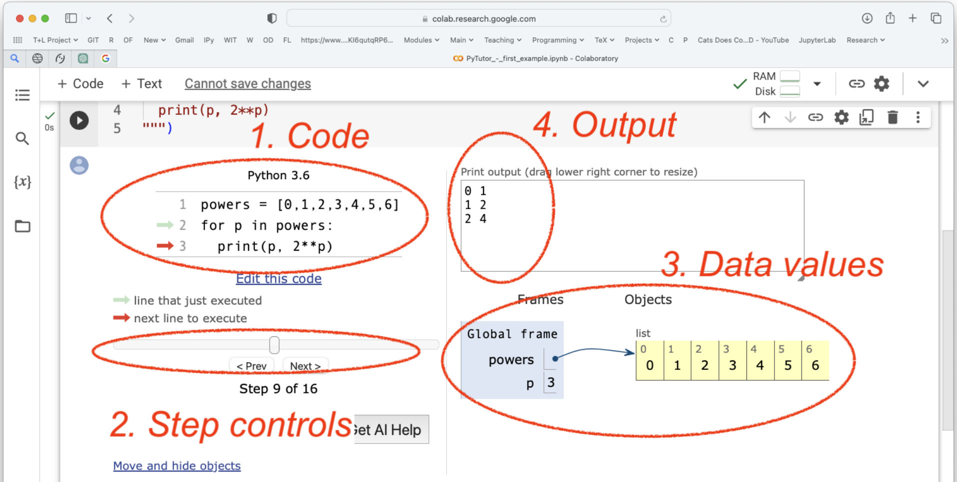
Step 4 — Click on second cell to run code in PyTutor



Using PyTutor with Colab

V

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



Outline

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*

```python
# Solution to Euler problem 2

# Calculate the sum of the even-values in the Fibonacci sequence
#    1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
# value that do not exceed four million,

last = 1
current = 2

answer = 0
while current <= 4_000_000:
    if current % 2 == 0:
        answer += current
    last, current = current, last + current

print(answer)
```

*This is a solution to the Euler 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*

```
# Solution to Euler problem

# Calculate the sum of the e
#    1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
# value that do not exceed four million,

last = 1
current = 2

answer = 0
while current <= 4_000_000:
    if current % 2 == 0:
        answer += current
    last, current = current, last + current

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, 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*

```
# Solution to Euler problem 2

# Calculate the sum of the
#    1, 2, 3, 5, 8, 13, 21,
# value that do not exceed

last = 1
current = 2

answer = 0
while current <= 4_000_000:
    if current % 2 == 0:
        answer += current
    last, current = current, last + current

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, at the programming competition site, projecteuler.net.

First Look at Python Code

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.

To get an idea of Python,

```
1  # Solution to Euler probl
2
3  # Calculate the sum of th
4  #    1, 2, 3, 5, 8, 13, 21
5  # value that do not excee
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, 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*

```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)
```

Python has lots of little features that make coding nicer.

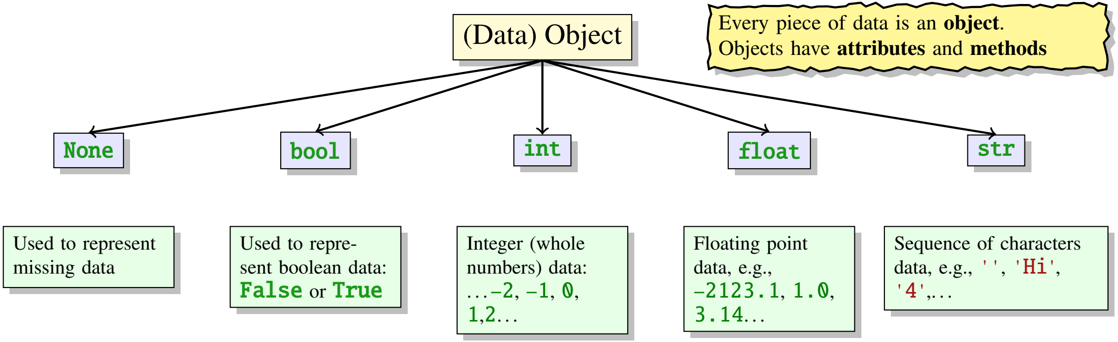
For example:

- 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, 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 II

```python
w = None

x = 4
y = '4'
z = 4.0

print(type(w), type(x), type(y), type(z))

x = x * 10
y = y * 10
z = z * 10

x = x * 1_000_000_000
z = z * 1_000_000_000

x = x / 1_000_000_000
z = z / 1_000_000_000

print(type(w), type(x), type(y), type(z))
```

Data and Data Types II

```
w = None

x = 4
y = '4'
z = 4.0

print(type(w), type(x), type(y), type(z))

x = x * 10
y = y * 10
z = z * 10

x = x * 1_000_000_000
z = z * 1_000_000_000

x = x / 1_000_000_000
z = z / 1_000_000_000

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 II

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

```python
w = None

x = 4
y = '4'
z = 4.0

print(type(w), type(x), type(y), type(z))

x = x * 10
y = y * 10
z = z * 10

x = x * 1_000_000_000
z = z * 1_000_000_000

x = x / 1_000_000_000
z = z / 1_000_000_000

print(type(w), type(x), type(y), type(z))
```

Data and Data Types II

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

```python
w = None

x = 4
y = '4'
z = 4.0

print(type(w), type(x), type(y), type(z))

x = x * 10
y = y * 10
z = z * 10

x = x * 1_000_000_000
z = z * 1_000_000_000

x = x / 1_000_000_000
z = z / 1_000_000_000

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

```python
z = set() # creating a empty set

# defining sets by stating values
a = {1,3,1,2,1,5,4}
b = {1,'a',3}

print(len(a)) # size of set

c = a & b # intersection
print(c)

c = a | b # union
print(c)

c = a - b # set difference
print(c)

c = b - a # set difference
print(c)
```

Collections: `set`

```python
z = set() # creating a empty se

# defining sets by stating values
a = {1,3,1,2,1,5,4}
b = {1,'a',3}

print(len(a)) # size of set

c = a & b # intersection
print(c)

c = a | b # union
print(c)

c = a - b # set difference
print(c)

c = b - a # set difference
print(c)
```

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

Collections: `set`

```python
z = set() # creating a empty set

# defining sets by stating values
a = {1,3,1,2,1,5,4}
b = {1,'a',3}

print(len(a)) # size of set

c = a & b # intersection
print(c)

c = a | b # union
print(c)

c = a - b # set difference
print(c)

c = b - a # set difference
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

```python
z = set() # creating a empty set

# defining sets by stating values
a = {1,3,1,2,1,5,4}
b = {1,'a',3}

print(len(a)) # size of set

c = a & b # intersection
print(c)

c = a | b # union
print(c)

c = a - b # set difference
print(c)

c = b - a # set difference
print(c)
```

Collections: list


```python
z = []  # creating a empty list

# defining lists by stating values
a = [1,3,1,2,1,5,4]
b = [1,3]

print(len(a)) # size of list

c = a + b # appending lists
print(c)

value = c[2] # list indexing ZERO-BASED
print(value)

d = c[2:5] # slicing SEMI-OPEN notation
print(d)

value = c[-4] # negative indexing
print(value)
```

Collections: `list`

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

```
z = []  # creating a empty list

# defining lists by stating values
a = [1,3,1,2,1,5,4]
b = [1,3]

print(len(a)) # size of list

c = a + b # appending lists
print(c)

value = c[2] # list indexing ZERO-BASED
print(value)

d = c[2:5] # slicing SEMI-OPEN notation
print(d)

value = c[-4] # negative indexing
print(value)
```

Collections: list

```python
z = []  # creating a empty list

# defining lists by stating values
a = [1,3,1,2,1,5,4]
b = [1,3]

print(len(a)) # size of lis

c = a + b # appending lists
print(c)

value = c[2] # list indexing
print(value)

d = c[2:5] # slicing SEMI-OPEN notation
print(d)

value = c[-4] # negative indexing
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`

```
z = []  # creating a empty list

# defining lists by stating values
a = [1,3,1,2,1,5,4]
b = [1,3]

print(len(a)) # size of

c = a + b # appending lis
print(c)

value = c[2] # list inde
print(value)

d = c[2:5] # slicing SEMI-OPEN notation
print(d)

value = c[-4] # negative indexing
print(value)
```

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

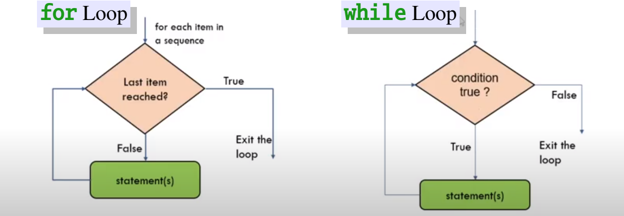
Because data values in a `list` have position (**index**) we can access particular value(s) using

- **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

```python
# for loops runs over a collection

print('Looping over a list')
for letter in ['a', 'e', 'i', 'o', 'u']:
    print(letter, 'is a vowel')

# BUT be careful if the collection is a set
# since a set does not have order
print('Looping over a set')
for letter in {'a', 'e', 'i', 'o', 'u'}:
    print(letter, 'is a vowel')

# Function range is useful in creating collections
# NOTE Python uses SEMI-OPEN intervals !!!
print('Use range to build collections')
for x in range(5):
    print(x)
```

Looping: `for` Loop

```python
# for loops runs over a colle

print('Looping over a list')
for letter in ['a', 'e', 'i', 'o', 'u']:
    print(letter, 'is a vowel')

# BUT be careful if the collection is a set
# since a set does not have order
print('Looping over a set')
for letter in {'a', 'e', 'i', 'o', 'u'}:
    print(letter, 'is a vowel')

# Function range is useful in creating collections
# NOTE Python uses SEMI-OPEN intervals !!!
print('Use range to build collections')
for x in range(5):
    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

```
# for loops runs over a c
print('Looping over a li
for letter in ['a', 'e', '
    print(letter, 'is a v

# BUT be careful if the c
# since a set does not ha
print('Looping over a se
for letter in {'a', 'e', '
    print(letter, 'is a vowel')

# Function range is useful in creating collections
# NOTE Python uses SEMI-OPEN intervals !!!
print('Use range to build collections')
for x in range(5):
    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

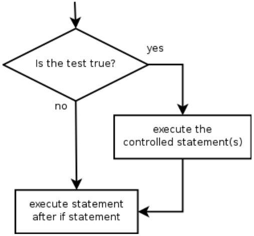
`range(START, END, STEP)`

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

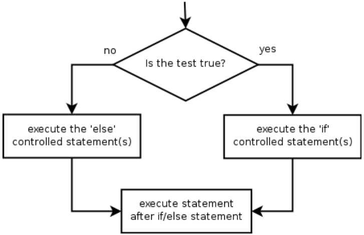
Making Decisions: `if`, `elif`, `else`

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

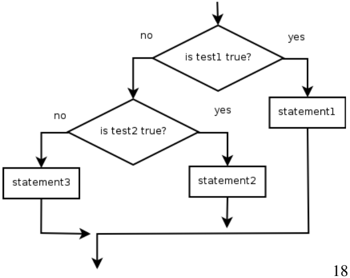
```
if test:
    statements
```




```
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`

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

for k in range(1,21):
    if k%15==0:  # is k a multiple of 15?
        print("fuzz buzz")
    elif k%3==0: # is k a multiple of 3?
        print("fuzz")
    elif k%5==0: # is k a multiple of 5?
        print("buzz")
    else:
        print(k)
```

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

```python
# In the drinking game of fuz
# but replace multiples of 3
# multiples of 5 with 'buzz'
# and multiples of 15 with 'I

for k in range(1,21):
    if k%15==0:  # is k a multiple of 15?
        print("fuzz buzz")
    elif k%3==0: # is k a multiple of 3?
        print("fuzz")
    elif k%5==0: # is k a multiple of 5?
        print("buzz")
    else:
        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`.

Function name

parameters

```python
def add(num1, num2):

    print("Number 1", num1)
    print("Number 2", num2)
    result = num1 + num2

    return result

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

function body

Return value

Function call