Python 2

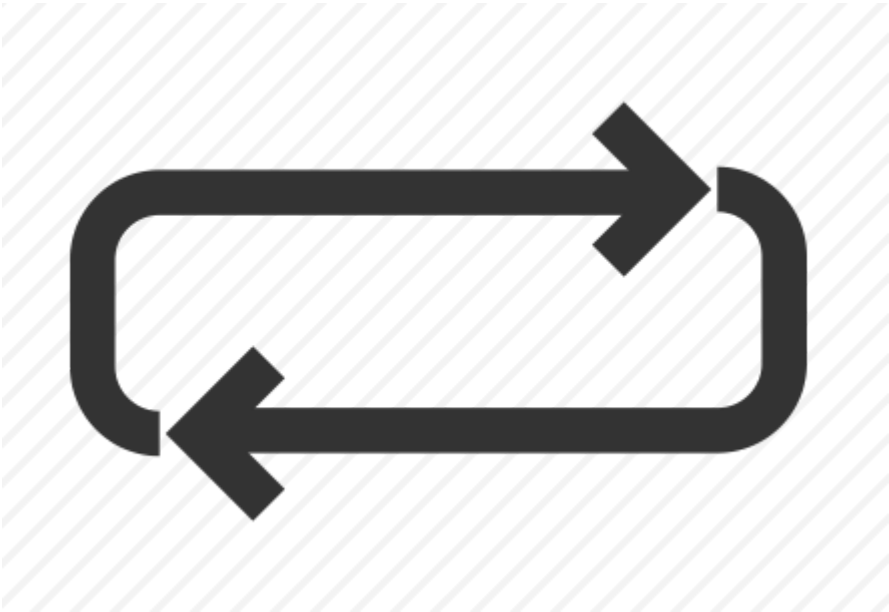
Overview

- Loops
- Dictionaries
- Exceptions
- Functions
- Googling!

Learning Objectives

- Understand and implement loops
- Understand and implement dictionaries
- Understand what exceptions are and how to handle them
- Understand and implement functions
- Understand good Googling practices

Loops



Loops

- Programming construct that allow us to **iterate** over a sequence of values
- Execute the same block of code a number of times
- Python has two built-in loop commands: *for* and *while*

Iterate meaning to repeat a process in order to generate an outcome

---

For Loops

Used when you want to repeat a block of code a **fixed** number of times

```
for x in [0,1,2]:
  print(f'current number is {x}')

# Output
# current number is 0
# current number is 1
# current number is 2
```

While Loops

We can execute a block of code as long as a condition is *true*:

```
i = 0

while i < 5:
  print(i)
  i += 1

# Output will be 0, 1, 2, 3, 4
```

Ask someone why it stops printing after 4.

It should be noted that the constructs such as continue, break and nesting can still be used with a while loop when shown in the next few slides.

Range

```
for x in range(0, 3):
  print(f'current number is {x}')
```

Returns a sequence of numbers in the specified range, which increments by 1 each time (by default)

Ask the class, why does it not print 3?

It's important to note that range generates values between the arguments, but not inclusive of the second argument. Value generated will be one less.

Demonstrate this.

You can also increment by more than 1 but don't need to worry about that right now.

Nested Loops

You can *nest* as many loops inside one another as you like:

```
adjectives = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for adj in adjectives:
  for fruit in fruits:
    print(adj, fruit)
```

Again ask the class what will happen here.

It's important to note that while you can nest as many as you like, it's good practise to limit that amount as the time it takes to execute the entire thing will take exponentially longer with each nested loop.

You'll also notice that we don't use range() here, if we use the in <-list-> the default is to loop through the entire list.

Break Keyword

With the *break* keyword we can stop the loop before it has looped through all of the items

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
  print(fruit)
  if fruit == "banana":
    break
```

Ask the class what will happen here.

Continue Keyword

With the *continue* keyword we can end the current iteration of the loop early and move to the next

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
  if fruit == "banana":
    continue
  print(fruit)
```

Ask the class what will happen here.

For Else

You can also use the keyword *else* to execute code after the loop is finished

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Ask the class what will be printed out.

You might notice that range() has only taken one argument, rather than two in previous slides.

When the range() function takes one argument the default starting index position is 0, the argument then specifies which position to stop iterating. In this case index position 6.

Why `for-else` exists, as opposed to just putting the `else:` code after the loop?

Answer: can `break;` out of the loop to prevent the `else:` from executing. Only useful in conjunction with `break` e.g:

```
some_list = ["item", "thing", "required"]

for i in some_list:
    if i == "required":
        break # We found what we were looking for
else:
    raise ValueError("Required not found.")

print("Required was found.")
```

https://stackoverflow.com/questions/9979970/why-does-python-use-else-after-for-and-while-loops

Quiz Time! 🤓

Given the below code, what are the range of values that will be printed?

```
for x in range(0, 5):
  print (x)
```

1. `0-4`
2. `0-5`
3. `1-4`
4. `1-5`

Answer: `1`

Given the below code, what are the range of values that will be printed?

```
i = 0
while i < 5:
  i += 1
  print(i)
```

1. 0-5
2. 1-5
3. 0-4
4. 1-4

Answer: 2

Exercise

Instructor to distribute exercises.

Dictionaries

Dictionaries

- A collection of values, similar to a list
- Used to store values as key-value pairs

```
car = {
  # key    # value
  'make': 'Jaguar',
  'model': 'XF',
  'year' : 2019,
  'isNew': False
```

```
}

print (car['make'], car['year'])
# Jaguar, 2019
```

Ask learners how they THINK a dictionary would be different to a list, how they would look up in one versus the other.

Python dictionaries work pretty much like a regular dictionary: you use a word (key) to lookup the meaning (value). Programmatic dictionaries come with a few more quirks but the comparison is mostly sound.

Make sure to denote the fact that you use x['y'] to access a key's value.

Dictionary Keys & Values

Keys

We use keys to unlock values, the most common key types are strings and numbers:

```
d = {
  'a': 'b',
  1: 'c',
  0.75: 'test'
  'd': [1, 2, 3]
  'e': { 'f': 'g' }
}
```

Values

Values can be pretty much anything! They can even be another dictionary

Keys must only be immutable (technically hash-able - they must have __hash__ method attached) which means you can use tuples and functions (for example) as dictionary key.

Dictionary Exceptions

If you try to access a value with a key that does not exist, an error will be raised

```
d = {'a': 1, 'b': 2}
>>> d['c']

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
```

Dictionary Mutations

Add an entry:

```
car['colour'] = 'Red'
```

Update an entry:

```
car['colour'] = 'Blue'
```

Delete an entry:

```
del car['colour']
```

Built-In Functions

What do these do?

```
>>> car.get(x)
```

Returns the value for a key if it exists

```
>>> car.items()
```

Returns a list of the key-value pairs

```
>>> car.clear()
```

Empties a dictionary

Ask the learners what they think each of these do.

N.B The get() function will either return:

1. The value for the specified key if key is in dictionary.
2. 'None' if the key is not found and value is not specified.
3. The optional second parameter value if the key is not found and value is specified.

`items()` is useful to get a list to iterate

Built-In Functions

```
>>> car.keys()
```

Returns a list of keys

```
>>> car.values()
```

Returns a list of values

Other useful things

```
# check if key exists
>>> 'colour' in car
True

# count how many keys there are
>>> len(car)
6
```

Quiz Time! 🤓

Given the below code, how would you access the *value* of the fruit's colour?

```
fruit = {
  'type': 'apple',
  'color': 'green'
}
```

1. `fruit['type']`
2. `color['green']`
3. `fruit['color']`
4. `fruit['color']['green']`

Answer: 3

Dictionary vs List

✅ Both mutable

✅ Both dynamic (can change in size)

✅ Can be nested

❌ List elements are accessed by position in list (indexed), dictionary elements are accessed by their keys

Nested: List can contain another list, dictionary can contain another dictionary. A dictionary can contain a list and vice versa.

What do you think will happen if we try to access the dictionary using an index number? Answer: KeyError

---

Dictionary vs List

When might you use a dictionary versus a list?

Scenario: You want to store the first name of every student in a class

- List

Scenario: You need to store detailed information about a specific vehicle, including make, model and colour

- Dictionary

Scenario: You need to store information about a number of different vehicles, and be able to look up each of those sets of information using the vehicle's number-plate

- Dictionary of dictionaries.

Student names:

- List probably sufficient because it's single-value items
- List allows duplicate values (More than one person may be called 'John' for example)
- How else could we handle duplicate values? (dictionary of Name:Count)

Detailed info:

- Why not a list? Because it is hard to access the info without a key into it, it would require searching the list for an item with particular values

List of detailed info:

- Why not a LIST of dictionaries? Could be but we'd need to search through the whole list to find the information
- Ask: Would it be possible to use driver name as a good way to index into the vehicle information? Is there any problem with this?

```
car1 = {
  'numberplate' : 'CP69 MAA',
  'brand': 'Jaguar',
  'model': 'XF',
  'color': 'Silver'
}
```

```
  car2 = {
    'numberplate' : 'MA59 AVD',
    'brand': 'Ford',
    'model': 'Mustang',
    'colour': 'Red'
  }

  cars = {
    'CP69 MAA' : car1,
    'MA59 AVD' : car2
  }
```

Note:

- Lists are ordered sets of objects (ordered by index, not alphabetically though they can be sorted) whereas dictionaries are unordered sets.
- Items in dictionaries are accessed via keys and not via their position.

So entirely depends on the use-case. If we need ordering, definitely lists Note: Ordered by insertion order, not by sorted order (but you can sort a list, you can't sort a dictionary) If we need to store duplicate values, also list (keys can't be duplicated) If we want to look-up values by their key then dict

---

Exercise

Instructor to distribute exercises.

Exceptions

Example

```
>>> numbers = [1,3,5,7,9]
>>> numbers[5]

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

What happened?

Ask the learners if they understand what the problem was.

Exceptions

An **exception** is an error from which your application cannot recover.

Exceptions are thrown when something unexpected happens in your program.

Examples?

What exceptions can the learners think of?

Examples in the following slides...

Types of Exceptions

Trying to change the type of some variable when it doesn't make sense, e.g. casting a string to a integer

```
>>> '2' + 2

TypeError: can only concatenate str (not "int") to str
```

Getting the sixth item from a list that only has 5 items

```
>>> numbers = [1,3,5,7,9]
>>> numbers[5]

IndexError: list index out of range
```

Types of Exceptions

Opening a file that doesn't exist

```
>>> f = open('does_not_exist.py')

FileNotFoundError:
  [Errno 2] No such file or directory: 'does_not_exist.py'
```

Dividing by zero:

```
>>> 1 / 0

ZeroDivisionError: division by zero
```

And more!

Handling Exceptions

Python has a built-in construct called `try-except` to handle exceptions.

The code we would normally run is placed inside the `try` block.

If an exception is raised inside the block, it immediately jumped to the `except` block to handle the exception.

```
try:
  x = 1 / 0
except:
  print("You can't do that!")
  print("We can still run the program after this though")

...
# More code carries on
```

A few pointers about exceptions

1. Minimise the amount of code you put in your exceptions, i.e. no try-except around your entire app, *just in case*.
2. Exceptions should be used **sparingly**, as they are no substitute for adequate input handling
3. When handling an exception, print out helpful messages explaining what happened and why in detail

1. You want to be as specific as possible with the type of exception you need to handle.
2. Input handling refers to checking the inputs of a function or passed to the program as args.
3. Helps us debug programs quicker!

Syntax errors vs Exceptions

Syntax errors occur when Python detects a statement has been written incorrectly:

```
>>> print('hello'))
  File "<stdin>", line 1
    print('hello'))
                  ^
SyntaxError: unmatched ')'
```

These are different to exceptions. Exceptions happen when syntactically correct code fails.

Other types of 'Error' ?

Quiz Time! 🤓

True or false: this code snippet will raise an exception.

```
stuff = [1, 2, 3, 4, 5, 6, 7, 8, 9]

try:
    for x in stuff:
        print (stuff[x + 1])
except:
    print ('Something went wrong!')
```

Answer: `True`

Exceptions as Variables

Inside the `except` block, you can contain the details of the exception in a variable:

```
try:
  x = 1 / 0
except Exception as e:
  print(e)
```

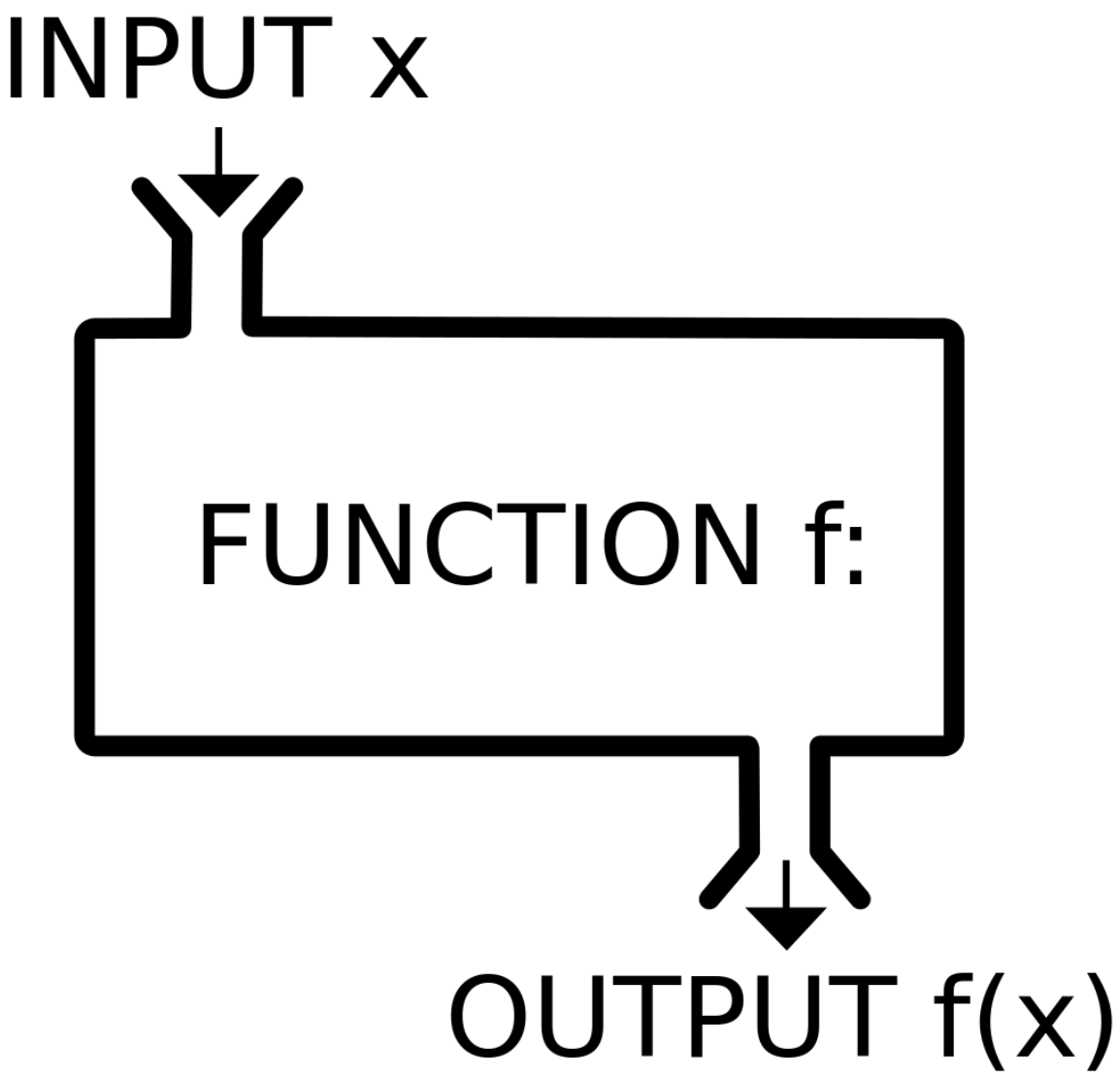
```
$ division by zero
```

Exercise

1. Write a small program that will purposefully throw an exception, causing your program to crash
2. Update it to include exception handling with a `try-except` block
3. Update it to print the error generated by the exception
4. Add an additional code path in your application which can generate another exception of of a different type and handle it seperately if it occurs
5. Add functionality to print out the stack trace for an exception to see where in code it occurs (Hint: `traceback`)

This one can be a fun code-along or ask a learner to show their code.

Functions



Functions

- Allow you to break up your program into a number of **reusable** chunks
- Functions work on variables that you pass it, which are known as arguments
- A function may take zero or more arguments
- Defined by the *def* keyword

```
# With arguments
def add_numbers(a, b):
  return a + b

# Without arguments
def get_name():
  return 'Alice'
```

Return Statement

You may have noticed the *return* keyword:

```
def get_name():
  return 'Alice'

name = get_name()
print(name) # Alice
```

- A function always has the option to return a value, but it doesn't have to return anything
- You can either use the `return` keyword without a value, or omit entirely

If you use `return` without a value the program returns a `none` value and continues to execute.

Calling a Function

Simply use the name of the function!

```
def add_numbers(a, b):
  print (a + b)

add_numbers(1, 2)
# 3
```

Function Arguments

- Information can be passed to a function as *arguments*
- Arguments are the variable names specified after the function name
- You can have as many as you need

```
def my_function(a, b, c):
  print(a, b, c)

my_function('hello', 'world', '!')
# hello world !
```

Again it's important to note that while you can have as many arguments as you want, it's always good practice to limit the amount you use.

Quiz Time! 🤓

Given the below code, how would you call this function and store the result in a variable?

```
def add_numbers(a, b, c):
    return a + b + c
```

1. `add_numbers()`
2. `add_numbers(1, 2, 3)`
3. `result = add_numbers()`
4. `result = add_numbers(1, 2, 3)`

Answer: `4`

Arbitrary Arguments

You can call a function with many different arguments using `*args`.

Effectively, `*args` creates a list of arguments.

```
def my_function(*people):
  for person in people:
    print(person)

my_function("Alice", "Bob", "John")
```

*args are useful when you don't know how many arguments you will be passed. E.g. the print function might receive varying amounts of arguments at different times e.g. print(1, 2, 3) could be print(1, 2, 3, 4, 5).

Arbitrary Arguments

There is a similar keyword `**kwargs` which accepts *named* arguments.

Effectively, `**kwargs` creates a dictionary of key-value arguments.

```
def concatenate(**kwargs):
  result = ""
  # Iterating over the Python kwargs dictionary
  for arg in kwargs.values():
    result += arg
  print (result)

concatenate(a="Real", b="Python", c="Is", d="Great", e="!")
```

`**kwargs` or Key Word Arguments Similar to `*args`, `**kwargs` are useful when you don't know how many arguments you will be passed. **kwargs have the added advantage that the arguments are named using key words. E.g. configuring a database connection to override any defaults. If there is a default

database name is Amazon-Database, you might add a kwarg of "db-name = My-Database" where it is a **kwarg in the function call.

Ordering of Arguments

Ordering of arguments is important in Python. The correct order is:

1. Standard arguments
2. *args arguments
3. **kwargs arguments

```
def my_function(a, b, *args, **kwargs)
```

Google-fu (or your search engine of choice)

Learning how to learn and adapt is just as important as remembering everything we're throwing at you here!

The best tool for this is Google, and knowing what to Google and how to interpret results is valuable

While there are jokes about software engineering just being "Googling stuff", doing so in the right way is difficult

---

Worked example

I, as a developer, want to know how to remove the item `Tractor` from the following list, but I don't know how

```
luxury_cars = ['Ferrari', 'Lamborghini', 'Tractor', 'Porsche']
```

Instructor demonstration: Show how you would Google removing an item from a list in Python, and how you would interpret the result - can potentially ask a learner to demonstrate on this or the next one. Get people participating with suggestions!

Errors!

Knowing how to Google is especially important for errors! Let's say I've run the following code and got the following error output:

```
func = "ThisIsNotAFunc"
print(func())
```

```
TypeError: 'str' object is not callable
```

Again, go through how you might solve this issue, focusing more on interpreting wordy results here

---

Exercise

Instructor to distribute exercises.

Learning Objectives Revisited

- Understand and implement loops
- Understand and implement dictionaries
- Understand what exceptions are and how to handle them
- Understand and implement functions
- Understand good Googling practices

Terms and Definitions Recap

Loop: Programming construct that allow us to iterate over a sequence of values

Dictionary: A collection of objects, similar to a list

Function: Allow you to break up your program into a number of reusable chunks, with each performing a particular task

Argument: A value that must be provided to the function

Iteration: The repetition of a process in order to generate an outcome

---

Further Reading

Loops

Dictionaries

Functions

*args and **kwargs explained