

Python 2

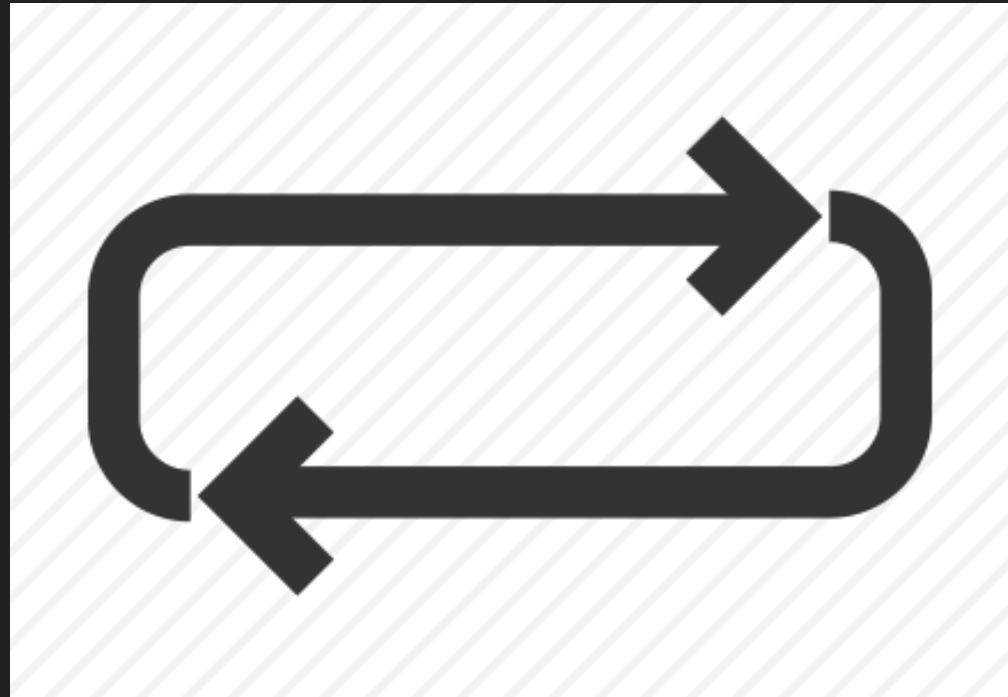
Overview

- 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*

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
```

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)

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)
```

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
```

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)
```

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!")
```

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  
    'brand': 'Ford',  
    'model': 'Mustang',  
    'year' : 1964,  
    'isNew': False  
}  
  
print (car['brand'], car['year'])  
# Ford, 1964
```

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

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

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

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.

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?

Exceptions

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

Exceptions are thrown when something unexpected happens in your program.

Examples?

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

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:

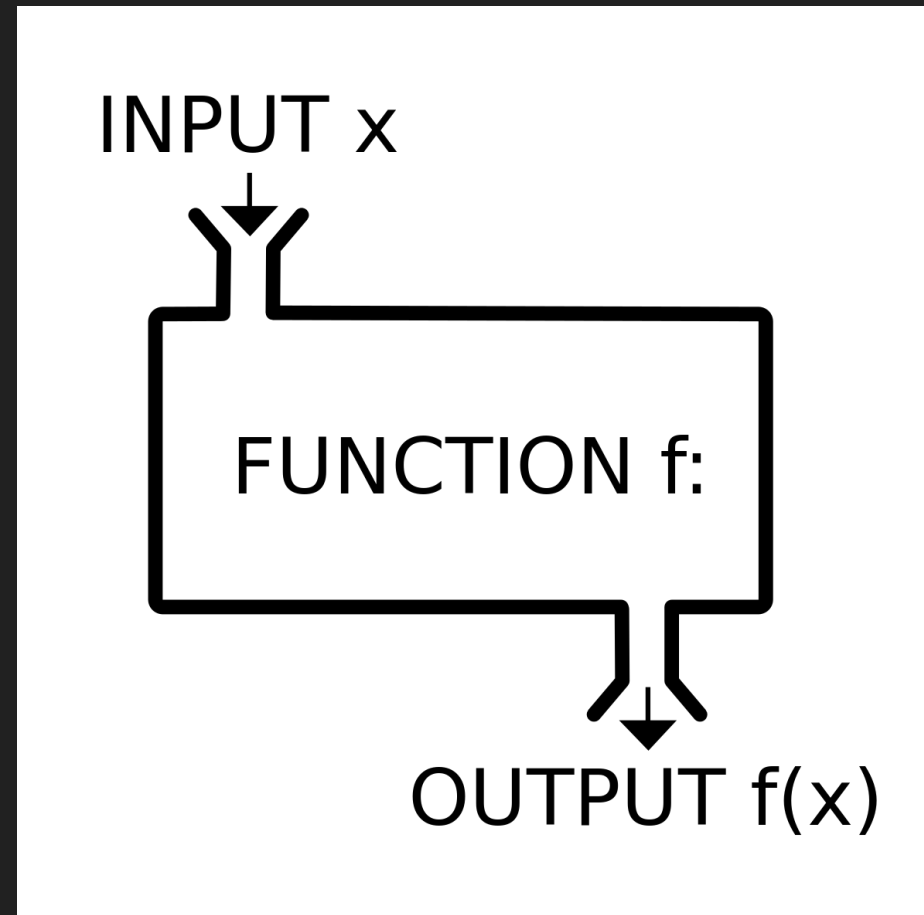
```
try:  
    x = 1 / 0  
except Exception as e:  
    print(e)
```

```
$ 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 a different type and handle it separately if it occurs
5. Add functionality to print out the stack trace for an exception to see where in code it occurs (Hint: `traceback`)

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

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 !
```

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")
```

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="!")
```


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

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']
```

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
```

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](#)