Coding Club

Code Cards

For KS3, GCSE and IGCSE

Python 3

Index

print()

The `print()` function is very flexible:

```
>>> print("Hello World")
Hello World
```

```
>>> print("Hello World, " * 3)
Hello World, Hello World, Hello World,
```

```
>>> print("Hello World\n" * 3)
Hello World
Hello World
Hello World
```

```
>>> text = "Hello"
>>> name = "Mia"
>>> print(text, name)
Hello Mia
```

```
>>> print(text, "your name is", name)
Hello your name is Mia
```

```
>>> print(text, "your name is", name, ".")
Hello your name is Mia .
```

```
>>> print(text, " your name is ", name, ".", sep="")
Hello your name is Mia.
```

```
>>> print(text, name, sep=",")
Hello,Mia
```

input()

The input() function is not quite as flexible. It can only accept a single string:

```
>>> name = input("What is your name? ")
What is your name? Daniel

>>> print(name)
Daniel
```

We can work around this though:

```
>>> my_name = "Mia"
>>> message = "My name is " + my_name
>>> message = message + ". What is your name? "
>>> user_name = input(message)
My name is Mia. What is your name? Daniel

>>> print(user_name)
Daniel
```

Ask the user for an integer:

```
user_age = int(input("Enter your age: "))
```

The input() function can be used to end a program nicely by providing this as the last line of code. Instead of suddenly ending, this waits for the user to end the program:

```
# exit nicely
input("\n\nPress RETURN to finish.")
```

"strings"

Coding Club

Strings can be added to and multiplied:

```
>>> my_string = "One ring to"
>>> my_string
'One ring to'

>>> my_string = my_string + " rule them all"
>>> my_string
'One ring to rule them all'

>>> my_string = my_string * 2
>>> my_string
'One ring to rule them allOne ring to rule them all'
```

Some escape sequences:

Escape sequence	What it does
\n	creates a line return in a string
\t	creates a tab style indent in a string
\\	allows a backslash to appear in string
\"	allows a speech mark to appear in a string

Single or double quotes? You choose – but be consistent:

```
>>> my_string = "\"Hi Mia,\" said Daniel."
>>> print(my_string)
"Hi Mia," said Daniel.
```

```
>>> my_string = '"Hi Mia," said Daniel.'
>>> print(my_string)
"Hi Mia," said Daniel.
```

mathematical operators

Python understands maths. This can be used in scripts or directly in interactive mode:

```
>>> 4*5
20
```

Here are some of the more useful operators:

Operator	Name	Example	Answer
*	multiply	2*3	6
/	divide (normal)	20/8	2.5
//	divide (integer)	20//8	2
%	modulus (remainder)	20%8	4
+	add	2+3	5
-	subtract	7-3	4
**	exponent (raise to)	4**2	16

comparison operators

Comparison operators are most often used in `if` statements:

```
if a > b:
    # do something
```

Operator	Name
==	equal to
!=	not equal to
>	greater than

Operator	Name
<	less than
>=	greater or equal to
<=	less or equal to

while loops

While loops continue looping through a block of code while a test is true:

```
>>> n = 0
>>> while n < 3:
        print(n)
        n = n+1

0
1
2
>>>
```

Sometimes you might make a mistake in your code and the while loop never becomes false. This is an **infinite loop** and can be stopped by pressing **CTRL-C**

Other times you may want to intentionally create an infinite loop that only stops when something happens, for example in a game. This can be achieved by creating a while loop that is `True` and using the `break` key word to allow your program to escape from the loop when you are ready:

```
while True:
    # code that runs until game is over goes here

    if [game over test] == True:
        break
```

for loops

The for loop is most useful for looping through items in a container data-type e.g.

```
>>> colours = ("Red", "Orange", "Yellow")
>>> for colour in colours:
        print(colour, end=" ")
Red Orange Yellow
```

range()

The range function takes three integer arguments:

```
range([start], [up to but not including], [steps])
```

starts from zero if omitted — required — only used with both other arguments

```
>>> for i in range(6):
        print(i, end=" ")
0 1 2 3 4 5
>>> for i in range(2,6):
        print(i, end=" ")
2 3 4 5
>>> for i in range(2,6,2):
        print(i, end=" ")
2 4
```

Putting it all together:

```
>>> colours = ("Red", "Orange", "Yellow")
>>> for i in range(1,2):
        print(i, colours[i])
1 Orange
```

if elif else

These control statements are used with logical operators to provide control in programs:

if

```
>>> my_number = 7
>>> if my_number > 5:
        print("My number is big.")

My number is big.
```

else

```
>>> my_number = 2
>>> if my_number > 5:
        print("My number is big.")
    else:
        print("My number is small.")

My number is small.
```

elif

```
>>> my_number = 7
>>> if my_number < 5:
        print("My number is small.")
    elif my_number < 10:
        print("My number is medium sized.")
    elif my_number < 100:
        print("My number is big.")
    else:
        print("My number is huge.")

My number is medium sized.
```

functions

Python has some built in functions:

```
print("This is my number:", number)
```

an argument — another argument

To make your own functions use the `def` key word:

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

function name — parameters

Calling the function:

```
add_two_numbers(3,4)
7
```

arguments

Parameter and *argument* are often used interchangeably. Strictly speaking, *parameters* are used in the function definition. When we call a function we pass it *arguments*.

Calling a function from the keyboard in tkinker (Card 11):

```
# bind up arrow to the move_up() function:
 window.bind("<Up>", move_up)
```

random numbers:

To use the random function, first import the random module:

```
import random
dice_number = random.randint(1,6)
```

turtle

First import the turtle module.
To avoid confusion between turtle commands and your own function names, it is often a good idea to import the turtle module and call its commands like this:

```
import turtle as t

t.forward(50)
```

Some great turtle commands:

Command	Arguments	Example
`forward() or fd()`	distance (pixels)	`forward(50)`
`back() or bk()`	distance (pixels)	`back(50)`
`right() or rt()`	angle (degrees)	`right(90)`
`left() or lt()`	angle (degrees)	`left(90)`
`home()`	none required (turtle to start)	
`penup()`	none required	
`pendown()`	none required	
`speed()`	10 = fast, 1 = slow, 6 = normal 0 = fast as possible	`speed(6)`
`pensize()`	line width (pixels)	`pensize(10)`
`pencolor()`	common colours	`pencolor("red")`
`shape()`	arrow, turtle, circle, square, triangle, classic	`shape("turtle")`

turtle continued

Some more turtle commands:

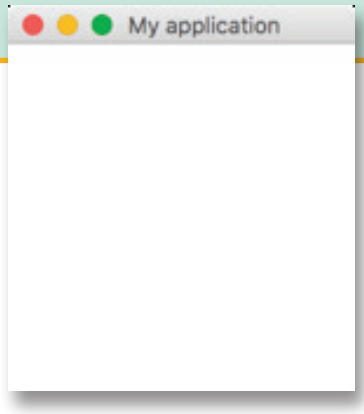
Command	Arguments	Example
`circle()`	*radius* (in pixels) *extent* - angle of circle to draw *steps* - number of lines used to make the circle (can make polygons)	`# Draw pentagon` `circle(50,steps=5)`
`fillcolor()`	common colours	`fillcolor("violet")`
`begin_fill()`	none required (creates a start point to fill a shape)	
`end_fill()`	none required (creates a stop point when filling a shape)	
`hideturtle()`	none required	
`showturtle()`	none required	
`color()`	common colours (turtle colour)	`color("brown")`
`setposition()`	*x* and *y* coords from the origin (pixels)	`setposition(50,60)`
`done()`	none required (tells Python to stop waiting for turtle commannds)	

see also: `https://docs.python.org/3.4/library/turtle.html`

tkinter basics

The tkinter package provides a simple windowing toolkit for your Python programs.

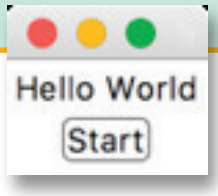
To create a simple, empty window:




```
from tkinter import *

# Create a window and add a title:
window = Tk()
window.title("My application")

# Other code goes here

# Start the infinite loop which watches for changes:
window.mainloop()
```

Laying out widgets:

Using the grid layout manager some sophisticated layouts can be achieved. Using `grid()` the window can be split into columns and rows starting from top left `(row=0, column=0)`.

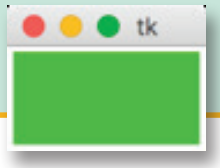



```
from tkinter import *
window = Tk()

def bye():
    my_label.config(text="Bye bye")

my_label = Label(window, text="Hello World")
my_label.grid(row=0, column=0)

my_button = Button(window, text="Start", command=bye)
my_button.grid(row=1, column=0)

window.mainloop()
```

tkinter widgets

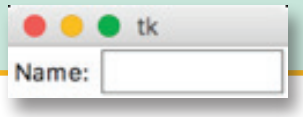
On card 11 there was a **button** and a **label**. Here are some other useful widgets that can be added after `window = Tk()`

Add a **canvas:**



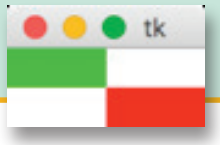

```
my_canvas = Canvas(bg="green", height=50, width=100)
my_canvas.grid(row=0, column=0)
```

Add a **text entry box** with a **label**:



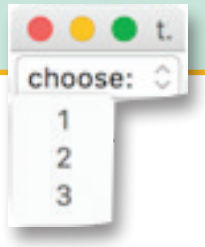

```
Label(window, text="Name:").grid(row=0, column=0)
my_text_box = Entry(window, width=10)
my_text_box.grid(row=0, column=1)
```

Add a **frame**:




```
frame1 = Frame(window,height=20,width=50,bg="green")
frame1.grid(row=0, column=0)
frame2 = Frame(window,height=20,width=50,bg="red")
frame2.grid(row=1, column=1)
```

A recipe to add a **drop-down menu**:




```
options = (1,2,3)
var = IntVar()
var.set("choose:")
my_dropdown = OptionMenu(window, var, *options)
dropdown.grid()
```

using images

First, images need to be loaded into memory:

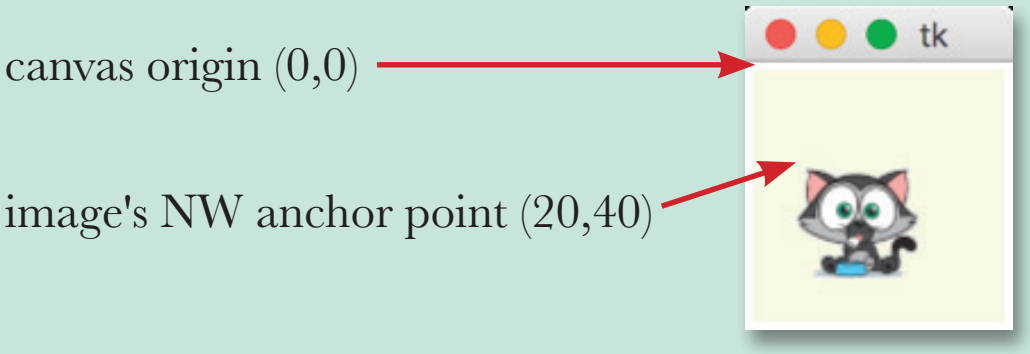
```
cat = PhotoImage(file="images/cat.gif")
```

Images can be added to buttons (see card 11):

```
my_button = Button(window, image=cat)
```

Images can be added to a tkinter canvas (see card 12):

```
from tkinter import *
window=Tk()

# build a canvas:
canvas = Canvas(window, bg="beige", height=100,
                width=100)
canvas.grid(row=0,column=0)

# add an image:
cat = PhotoImage(file="images/cat.gif")
canvas.create_image(20, 40, image=cat, anchor=NW)

window.mainloop()
```



importing modules

Modules are files or groups of files outside of your own script.

Importing a big group of modules the easy way:

```
from tkinter import *
```

Your code can now call the methods and functions directly from the tkinter module without identifying where the code is from:

```
Label(window, text="Name:")
```

Never do this more than once in your code as you can start to confuse where the functions and methods come from.

Importing a module and keeping track of where the functions are coming from:

```
import turtle
```

More typing is required but it is clear which code belongs with which module:

```
turtle.forward(50)
```

The best of both worlds:

```
import turtle as t
```

It is still clear where the functions come from but less typing is needed:

```
t.forward(50)
```

tuples

Tuples are the simplest and most memory efficient container data-type available in Python. They are used to store a group of elements that will not change:

Tuples are created with **round brackets**:

```
>>> my_tuple = ("Mon", "Tue", "Wed", "Thu", "Fri")
>>> my_tuple
('Mon', 'Tue', 'Wed', 'Thu', 'Fri')
```

Find the length of a tuple (how many items it contains)

```
>>> len(my_tuple)
5
```

The elements are indexed from 0:

```
>>> my_tuple[0]
'Mon'
>>> my_tuple[2]
'Wed'
```

Locate an element in a tuple:

```
>>> my_tuple.index("Thu")
3
```

lists

Lists can do everything a tuple can do and more. They are used to store a group of elements that can change:

Lists are created with **square brackets**:

```
>>> my_list = ["Mon", "Tue", "Wed", "Thu", "Fri"]
>>> my_list
['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
>>> len(my_list)
5
>>> my_list[2]
'Wed'
>>> my_list.index("Thu")
3
```

Lists can be combined:

```
>>> weekend = ["Sat", "Sun"]
>>> week = my_list + weekend
>>> week
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

Lists can be added to:

```
>>> my_list.append(3)
>>> my_list
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 3]
```

Elements can also be deleted and replaced in a list:

```
>>> del my_list[5]
>>> my_list[3] = "January"
>>> my_list
['Mon', 'Tue', 'Wed', 'January', 'Fri']
```

dictionaries

Dictionaries are another container data-type but here we define the key. It is best to think of dictionaries as unordered `key:value` pairs:

Dictionaries are created with **curly brackets**:

```
>>> my_dict = {1:"b", 2:"a", 3:"r"}
>>> my_dict
{1: 'b', 2: 'a', 3: 'r'}
>>> my_other_dict = {"red":2, "green":5, "blue":3}
>>> my_other_dict
{'green': 5, 'blue': 3, 'red': 2}
```

Adding, replacing, deleting items and finding the number of elements in a dictionary is similar to lists:

```
>>> my_dict[5] = "t"
>>> my_dict
{1: 'b', 2: 'a', 3: 'r', 5:'t'}
>>> my_dict[5] = "g"
>>> del my_dict[3]
>>> my_dict
{1: 'b', 2: 'a', 5:'g'}
>>> len(my_dict)
3
```

Extracting keys and values into their own lists:

```
>>> keys_list = list(my_dict.keys())
>>> keys_list
[1, 2, 5]
>>> values_list = list(my_dict.values())
>>> values_list
['b','a','g']
```

string manipulation

Strings can be treated like a container data-type:

Concatenation (adding strings):

```
>>> string1 = ("Hello")
>>> string2 = ("World")
>>> string3 = string1 + " " + string2
>>> string3
'Hello World'
```

Find the number of letters:

```
>>> len(string3)
11
```

Find and replace letters:

```
>>> "e" in string3
True
>>> string3.find("l")  #Only finds first instance
2
>>> string3.count("l")
3
>>> string3[1]
'e'
>>> string3.replace("e", "a")
'Hallo World'
```

Convert to uppercase or lowercase:

```
>>> string4 = string3.upper()
>>> string5 = string3.lower()
>>> print(string3, string4, string5)
Hallo World HALLO WORLD hallo world
```

reading text files

Store a reference to a file in a variable:

```
my_file = open("my_doc.txt", "r", encoding="utf-8")
```

(This assumes the text file is in the same folder as the script.)

Store every line of text from your file in a list:

```
my_list = list(my_file)
```

Loop through this file a line or word at a time:

```
word_count=0
for line in my_file:
    words = line.split()
    for word in words:
        word_count = word_count+1
print(word_count)
```

Read a text file and store its contents in a string variable:

```
my_string = open("my_document.txt").read()
```

When finished, close the file to save system resources:

```
my_file.close()
```

writing to text files

This will create a new text file called `hi.txt`

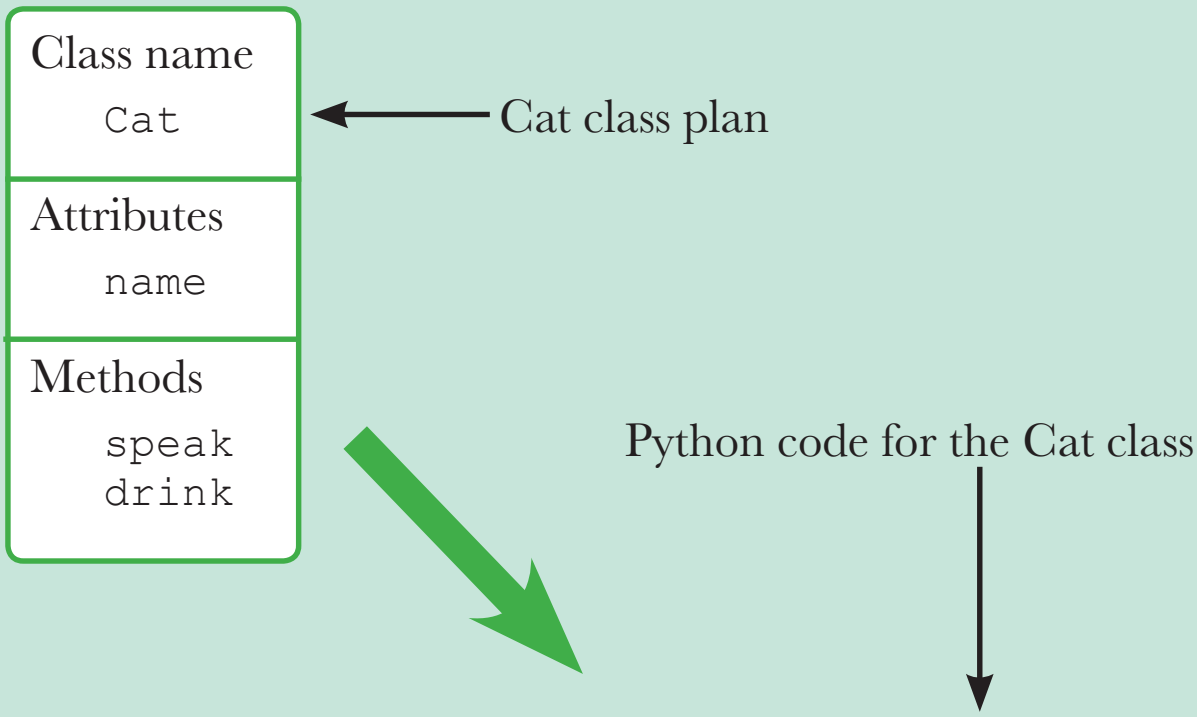
```
my_file = open("hi.txt", "w", encoding="utf-8")
my_file.write("Ça va\n")
my_file.write("André")
my_file.close()
```

needed for non-ASCII characters

classes

Classes are like factories. Many objects can be built from then when they are sent orders.

Classes have to be built carefully:




```
class Cat:
    # constructor:
    def __init__(self, name):
        self.name = name

    # methods:
    def speak(self):
        print(self.name, "says: 'Meow'")

    def drink(self):
        print(self.name, "drinks some milk")
        print(self.name, "takes a nap\n")
```

objects

(See card 20 for the corresponding Cat class)

From one class it is easy to create many different objects :

```
# Create two instances of a cat
romeo = Cat("Romeo")
juliet = Cat("Juliet")
```

Objects can call all the methods from their creator class:

```
# Play with Romeo
romeo.speak()
romeo.drink()

# Play with Juliet
juliet.speak()
juliet.drink()
```

Here is the output from playing with the cats:

```
Romeo says: 'Meow'
Romeo drinks some milk
Romeo takes a nap

Juliet says: 'Meow'
Juliet drinks some milk
Juliet takes a nap
```

Code Cards:

Code Cards provide indexed, quick reminders and recipes for the Python 3 code required at KS3, GCSE and IGCSE.
Keep them in your school jacket pocket (but remember to leave them behind during exams!)

Red cards cover material found in Coding Club Level 1 books.
Blue cards cover material found in Coding Club Level 2 books.
Green cards cover Level 3 topics, usually not required at GCSE.

These cards are available in a variety of formats.
Visit `www.codingclub.co.uk/codecards` for further details.

The Coding Club series for KS3:

Available from:
`http://education.cambridge.org` or Amazon.