# Introduction to Computational Physics (PHYS105)

## Lecture 4: Flow Control

Carl Gwilliam

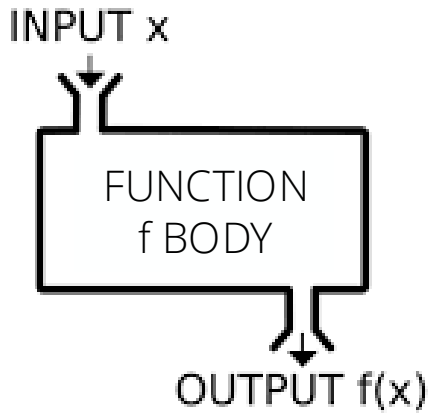
(C.Gwilliam@liverpool.ac.uk)


## Attendance code: 224329

# Lecture 3: Recap

- Last week we learnt how to define our own functions
  - Allowing us to write reusable calculations
  - Please write your code in functions where appropriate!

```
def function(input):
    # This is the body
    return output
```




- They consist of 3 main pieces
  - Inputs: passed as comma-separated set of arguments
  - Body: code that performs the work
    - Must be indented
  - Outputs: comma-separated values specified by `return`

[1]:
```
# <!-- Student -->
def capacitance(area, separation = 0.01, kappa = 1.):
    '''Calculate the capacitance of a parallel plate capacitor
    with given area and separation (in m), with an inter-plate
    matrial of relative permitivity kappa.
    '''
    epsilon0 = 8.85e-12
    return kappa * epsilon0 * area / separation

c = capacitance(2, kappa = 80.2)
print(c)
```

```
1.41954e-07
```

- We also looked more at plotting
  - Histograms, lines, and points with error bars

# Lecture 3: Formative problem solutions

- Exercise 1: Using np for sine & natural log
  - And creating arrays with `linspace`

[353]:
```
import numpy as np

print("sin(77) =", np.sin(np.deg2rad(77)))
print("log(17.6) =",  np.log(17.6))

array, step = np.linspace(3.0, 7.0, 5, retstep = True)
print("linspace array from 3-7 with 5 entries: ", array)
print("Which has a step size of", step)
```

```
sin(77) = 0.9743700647852352
log(17.6) = 2.8678989020441064
linspace array from 3-7 with 5 entries:  [3. 4. 5. 6. 7.]
Which has a step size of 1.0
```

- Notes on Ex 1:
  - NumPy has a function to convert deg → rads
  - `retstep = True` gives the step from `linspace`
- Notes on Ex 2:
  - It is good practice to add docstrings to explain what the function should do
  - Remember, the variables don't need to have the same name inside & outside the function

- Exercise 2: Function to calculate prism values

[354]:
```
# Define the function
def rectPrismParams(width, length, height):
    '''
    Return volume, surface area and length of edges of
    rectangular prism given its width,length and height
    '''
    vol = width * length * height
    area = 2 * (width * length + width * height + length * height)
    edge = 4 * (width + length + height)
    return vol, area, edge

# Use the function
w = 0.3
l = 0.17
h = 0.25
V, A, s = rectPrismParams(w, l, h)

print("Rectangular prism width",w,"length",l,"height",h, "has:")
print("Volume",V)
print("Surface area",A)
print("Length of edges",s)
```

```
Rectangular prism width 0.3 length 0.17 height 0.25 has:
Volume 0.012750000000000001
Surface area 0.337
Length of edges 2.88
```

# Lecture 3: Formative problem solutions (2)

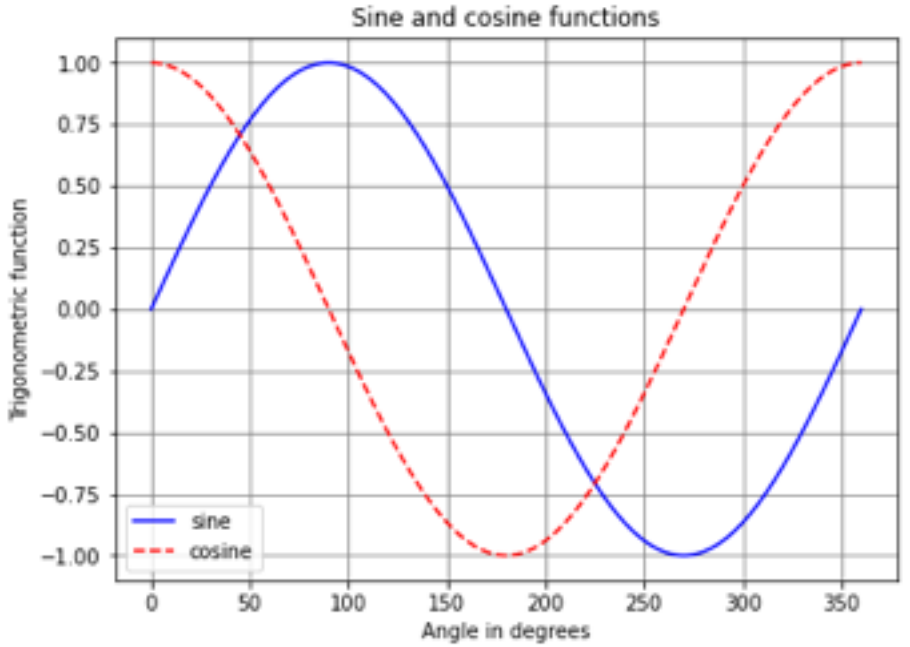
- Exercise 3: Use array-at-time functionality to take sine of angles from 0 to 360°
  - Can use `(a)round()` to set precision of a given array or set global print precision: `np.set_printoptions`

[377]:
```
# Create angles
angles_deg = np.linspace(0.0, 360.0, 361)
print("Angles (in deg) = \n",angles_deg, "\n")

# Take sine of array
np.set_printoptions(precision = 2)
sines = np.sin(np.deg2rad(angles_deg))
print("Sines = \n",sines)
```

```
Angles (in deg) =
 [  0.   1.   2. ... 358. 359. 360.]

Sines =
 [ 0.00e+00  1.75e-02  3.49e-02 ... -3.49e-02 -1.75e-02 -2.45e-16]
```

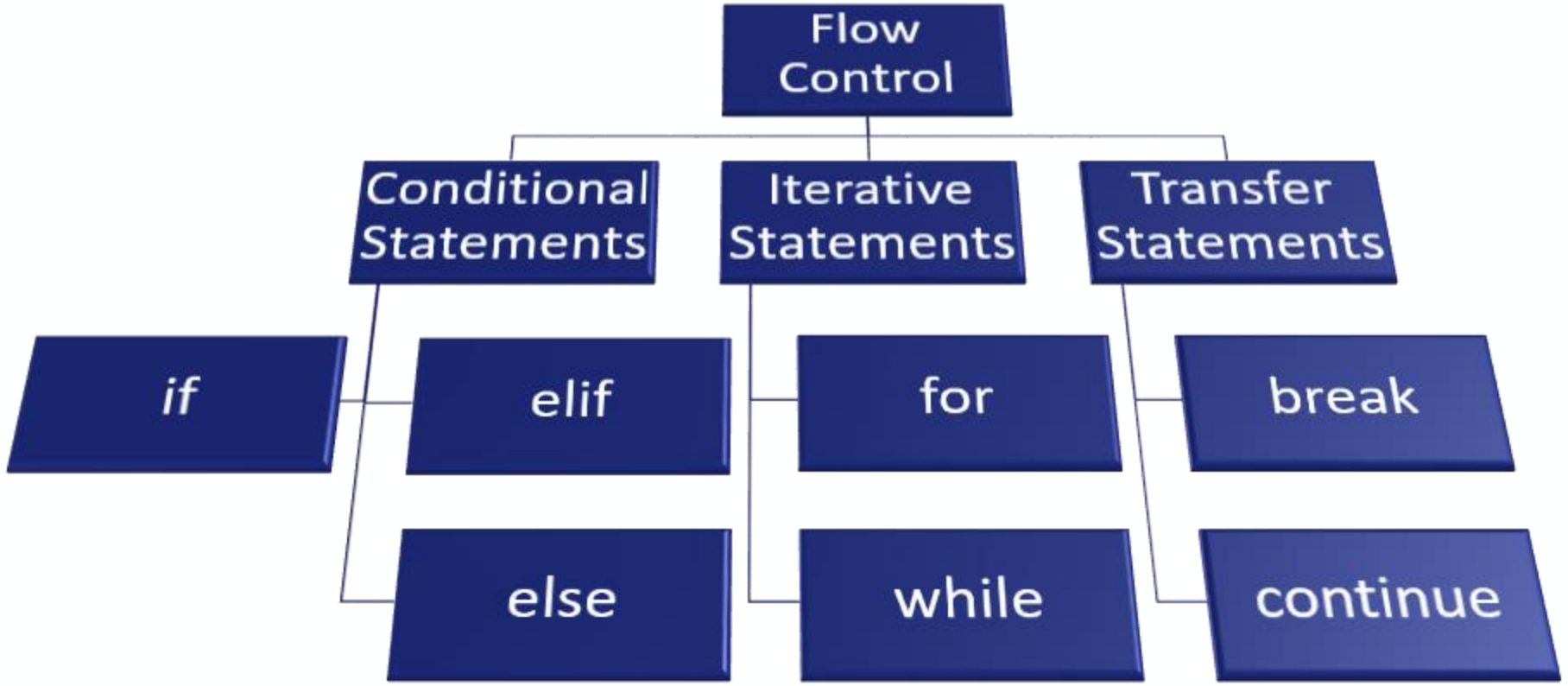
- Exercise 4: Plot sine and cosine functions
  - cos calculated just like sin above
  - Call `plot(x, y, …)` twice, once with each
    - Takes x and y array, not just y
  - Note: use different line colours and styles

[378]:
```
import matplotlib.pyplot as plt
%matplotlib inline

cosines = np.cos(np.deg2rad(angles_deg))

plt.figure(figsize = (7, 5))
plt.title("Sine and cosine functions")
plt.xlabel("Angle in degrees")
plt.ylabel("Trigonometric function")
plt.plot(angles_deg, sines, linestyle = '-',
         marker = '', color = 'b', label = 'sine')
plt.plot(angles_deg, cosines, linestyle = '--',
         marker = '', color = 'r', label = 'cosine')
plt.grid(color = 'grey')
plt.legend()
plt.show()
```



# Flow control

- We now have the structures to store data and the functions to operate on them
  - But the calculations we can do are limited since the code we have written so far is purely sequential




- Often, we want to execute code only if a certain condition is met (conditional) or we may want to repeat code multiple times (iterative) e.g. for each element of our data structure
  - Today, we will look at each of these in turn …

# Relational and boolean operators

- We can compare numerical values using relational operators: ==, !=, >=, <=, >, <
  - Similar to the corresponding mathematical symbol, but note == for equality (as = is assignment)

[379]:
```
# (In)Equality
print("2 == 2",2 == 2)
print("2 != 2",2 != 2)
```
```
2 == 2 True
2 != 2 False
```

[382]:
```
# Greater/less than
print("2 >  3",2 > 3)
print("2 <  3",2 < 3)
```
```
2 >  3 False
2 <  3 True
```

[381]:
```
# Greater/less than or equal
print("2 >= 2",2 >= 2)
print("3 <= 2",3 <= 2)
```
```
2 >= 2 True
3 <= 2 False
```

- The result of these is either `True` or `False`
  - This is represented by a special type of variable, called a boolean or `bool`

- Boolean values can be combined using logical, or boolean, operators: `not`, `and`, `or`
  - In the same way as e.g. logic gates in electronics

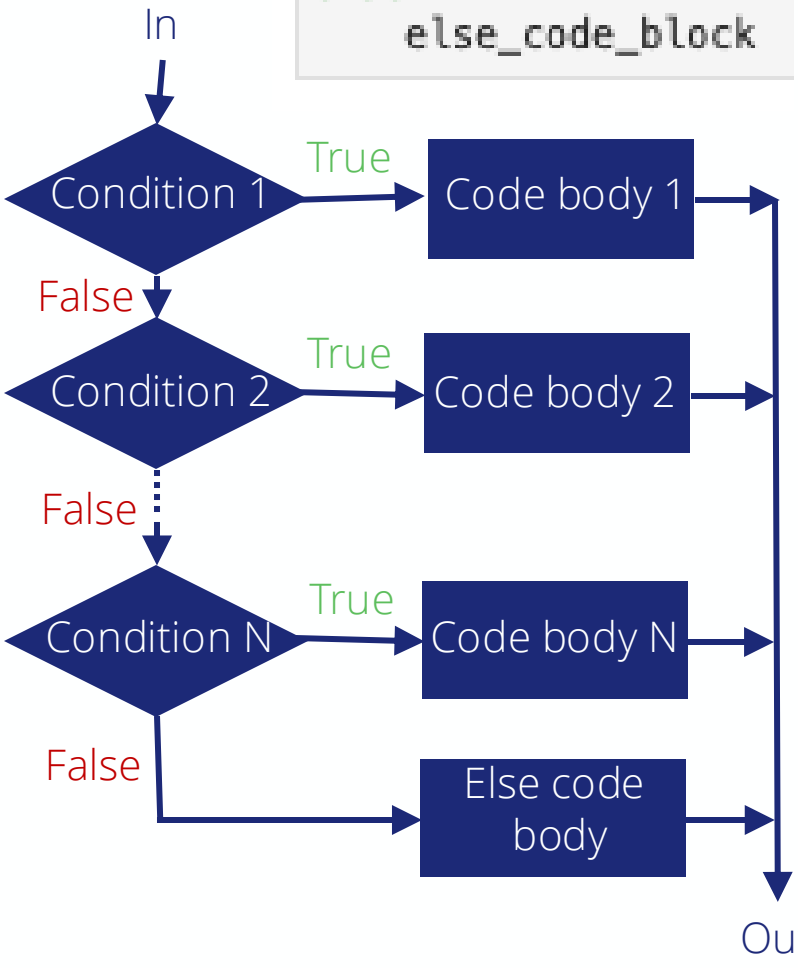
```
x, y = True, False
print("x and y =", x and y)
print("x or y =", x or y)
print("not x =", not x)
```
```
x and y = False
x or y = True
not x = False
```

| X | Y | X and Y | X or Y |
|---|---|---|---|
| True | True | True | True |
| True | False | False | True |
| False | True | False | True |
| False | False | False | False |

# Conditional execution

- Allows us to execute code depending on certain conditions
- Conditional execution is achieved via `if…elif…else`
  - The latter two are both optional
- Starts with `if` followed by a `bool` condition and a colon (`:`)
  - `True`: the indented code block is executed
  - `False`: passed to the next part of test if present
- Next, we have any potential `elif` conditions
  - Similar in format to the `if` condition
  - Only attempted if previous `if/elif` statement is `False`
- Finally, we may have an `else` statement
  - Has no associated condition
  - Executed if **all** the above `if/elif` statements are `False`

```
if condition_1:
    code_block_1
elif condition_2:
    code_block_2
elif condition_N:
    code_block_N
else:
    else_code_block
```



# Conditional execution (2)

- Let's look at some examples ....
  - Simple if condition:

[53]:
```
test = 0.5
if test < 1.0:
    print("test is less than 1")
```
```
test is less than 1
```

[55]:
```
test = 1.5
if test < 1.0:
    print("test is less than 1")
```

- Can also nest if...elif...else conditions:

```
test = 0.8
test2 = 'a'
if test < 1.0:
    print("test is less than 1")
    if test2 == 'a':
        print("... and test2 == a")
    else:
        print("... and test2 != a")
```
```
test is less than 1
... and test2 == a
```

- Full if...elif...else condition:

[6]:
```
print ("Checking test value", test)

if test < 1.0:
    print("... test is less than 1")
elif test < 2.0:
    print("... test is betewwn 1 and 2")
elif test < 3.0:
    print("... test is betewwn 2 and 3")
else:
    print("... test is greater than 3")

print("End of check")
```

```
Checking test value 0.5
... test is less than 1
End of check

Checking test value 2.3
... test is betewwn 2 and 3
End of check

Checking test value 5
... test is greater than 3
End of check
```

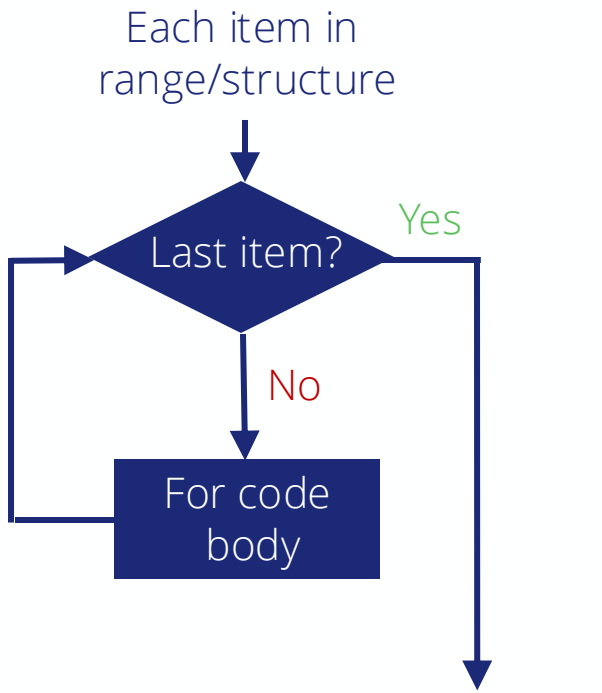
# Iterating

- Often want to execute code for a range of values
  - E.g. each item in a data structure (list, tuple, np.array, etc)

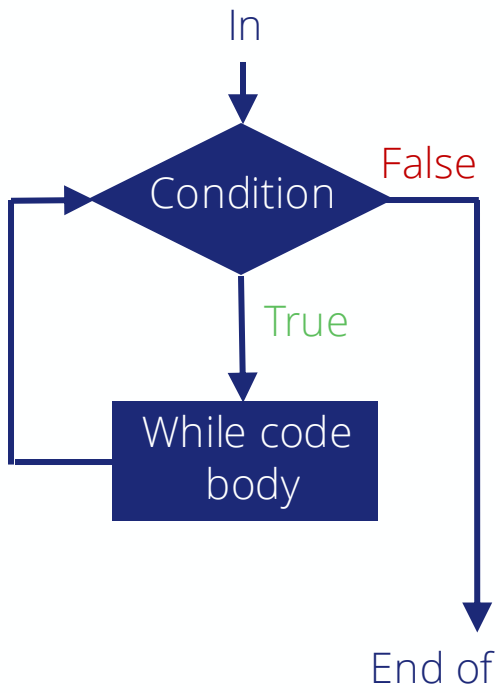
```
mylist = [1,2,3,...]
mylist[0] = mylist[0]**2
mylist[1] = mylist[1]**2
mylist[2] = mylist[2]**2
...
```

- Copy-and-pasting code is not scalable
  - Need a way of iterating over a range of values …

- For loop
  - Loop over a given range of values
  - Loop over each item in a data structure




- While loop
  - Iterate while a condition is true




- Let's look at each in turn …

# For loop: range

- The start of the loop is indicated by the `for` keyword
  - Followed by the name of a variable used as an index, which takes the variables given by the `range` function
    - Both the start and step are optional, defaulting to 0 and 1
  - The definition is ended by a colon (`:`)

```
for n in range(start, stop, step):
    code_block
```

- The indented code block is executed for each index value returned by range …

```
[66]: start = 2
      stop = 10
      step = 2
      for n in range(start, stop, step):
          print("Value squared",n**2)

      print("Final value of index",n)
```

Exclusive

```
Value squared 4
Value squared 16
Value squared 36
Value squared 64
Final value of index 8
```

- From this example we can see what happens:
  - The index `n` is set to start value of `range` i.e 2
  - The body of the for loop is executed
  - The value of the index `n` is incremented by step and compared to `stop`
    - If it is less than `stop` the body is executed
    - This continues in `step` increments
  - Until `n + step ≥ stop`, when the loop terminates
    - Final value of `n` is the last allowed value (e.g. 8)

# For loop: data structure

- For loops can also cycle directly through the elements of a data structure (list, tuple, np.array, etc)
  - Without the needed for an index

- The syntax here is very similar but without `range()`
  - And the variable after `for` now takes the value of the current element in the data structure not an index

```
for var in data_structure:
    code_body
```

- Here is an example:
  - The `var` is assigned to the first element ("one")
  - The body of the `for` loop is then executed
  - The `var` is assigned to each next item in the structure
    - And the body is executed
  - Until it has reached the end of the data structure

```
[67]: mylist = ["one", "two", "three"]

for var in mylist:
    print(var)

print("End of loop, var is", var)
```

```
one
two
three
End of loop, var is three
```

- Note:
  - The two forms of the for loop are not really different: in the first case the `range` function just returns a sequence of integers (kind of like a list) that the for loop iterates over in the same way as the second

# While loop

- A while loop provides another way to iterate, while a condition is true rather than over a sequence

- The start of the loop is indicated by the `while` keyword
  - Followed by a condition on a predefined variable
  - As always, the definition ends with a colon (`:`)

```
while condition:
    code_body
    # Body must change condition
```

- The indented code block is executed while the condition is true
  - Unlike the for loop, the user is responsible for updating the variable driving the condition. Else you'll get an infinite loop!

If this happens in the PC class, just restart the jupyter kernel .

- Let's see an example:
  - The condition is evaluated with initial value of `test` (0.3)
  - If the condition passes the body is executed
    - Unlike a `for` loop, body may never be executed
  - The body updates the value of the `test` variable
    - If the condition is still true body is executed again
  - Until the condition fails

[68]:
```
test = 0.3
limit = 1.1
step = 0.25

while test < limit:
    print("test =",test)
    test = test + step

print("Final value of test is", test)
```

```
test = 0.3
test = 0.55
test = 0.8
test = 1.05
Final value of test is 1.3
```

# Transfer: continue and break

- The `for` and `while` loops repeat a block of code until end of the range/list or the condition fails
  - But sometimes we may wish to alter this and there are two ways we can do this …

- The `continue` statement
  - Skip to beginning of next iteration
  - Without executing any further code in the current iteration

[83]:
```
numbers = [2, 736, 100, -6, 999]
for n in numbers:
    if n < 0:
        print("Skipping -ve number")
        continue
    print("Sqrt of", n, "=", math.sqrt(n))
```

```
Sqrt of 2 = 1.414
Sqrt of 736 = 27.129
Sqrt of 100 = 10.000
Skipping -ve number
Sqrt of 999 = 31.607
```

- The `break` statement
  - Jump directly to the very end of the loop
  - Without executing any further code in the current iteration nor any further iterations

[78]:
```
string = "This is a test"
for s in string:
    if s in ["a", "e", "i", "o", "u"]:
        break

print("First vowel is", s)
```

```
First vowel is i
```

- Note: here we have a new use of the `in` operator
  - As well as iterating in for loops it can be used to test if something is contained in a data structure

# Summary

- This week we looked at how to control the flow of our python program's execution
- How to only execute code in certain cases
  - `if … elif … else`
- How to repeatedly execute pieces of code
  - Over a range or data structure
  - While a condition is true
- This will allow us to write much more useful programs
- You will look at these further in the problems classes
- We only have one more week of building up our python
  - Then we will move on to real physics uses to practice

```
# Rich formatting of output text
from rich import print as rprint

lectures = {
    2 : "Python basics",
    3 : "Data structures",
    4 : "Functions and graphs",
    5 : "Looping and Conditionals",
    6 : "Input, Output and Formatting",
    7 : "Fitting data",
    8 : "Good practice and debugging",
    9 : "Analytic and numeric calculations",
    10 : "Further numeric techniques",
    11 : "Monte Carlo methods",
    12 : "Python modules"
}

for week, title in lectures.items():
    if week == 5:
        rprint ("[bold green]" + title + "[/bold green]")
    elif week < 5:
        rprint ("[green]" + title + "[/green]")
    else:
        rprint ("[italic red]" + title + "[/italic red]")
```

```
Python basics
Data structures
Functions and graphs
Looping and Conditionals
Input, Output and Formatting
Fitting data
Good practice and debugging
Analytic and numeric calculations
Further numeric techniques
Monte Carlo methods
Python modules
```