Imperial College London

Python for beginners

Katerina Michalickova, Research Computing Service and Computational Methods Hub

October 29, 2018

Slides

http://tinyurl.com/ichpcclass

Why programming?

Computing is an integral part of research

Some of us write short scripts, some of us code whole projects, some of us just need a good understanding of what a software packages does

Regardless, programming (or understanding of) became a basic lab skill

Try to think about today's lesson not as much as Python lesson but programming lesson that uses Python examples

To a certain extend, all programming languages offer the same tools or building blocks

We'll try to understand the basic building blocks today and it's up to you to learn to decompose your problem to fit those blocks

Programming code is a language

Programming language a is a formal language with no ambiguity, low redundancy and high literalness (it means exactly what is says).

Another examples of a formal language are mathematical notation or representation of chemical structures.

Natural language (what we speak) is at the opposite end of the spectrum – high ambiguity, high redundancy (to make up for high ambiguity) and low literalness (full of idioms and metaphors).

In other words, computers do exactly what you tell them. Think of them as an employee with particular strength in precision and speed and weakness in empathy and inability to grasp the big picture.

Programming languages

There are many languages out there and at any given time, several of them are going to be very popular.

When choosing a language, consider:

- it is suited to the problem?
- is there an active community?
- is it any good for the job market?

Imperial College London

High level languages

```
int main()
{
  printf("Hello world!\n");
  return 0;
}
```

hello_world.c

- interpreted – Perl, PHP, Python
  - need an interpreter
  - portable
  - easy to modify
- hybrid (bytecode stage) – Java, Python
  - need an interpreter
  - portable
  - easy to modify
  - faster
- compiled – C, Fortran
  - need a compiler
  - platform specific executables
  - fast

Imperial College London

Low level languages

- bytecode
  the last portable stage
- assembly and machine codes
  architecture specific

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello world!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-11)"
        .section        .note.GNU-stack,"",@progbits
```

assebly of hello_world.c

Universal language?

All languages converge at the stage of machine code.

Making it work - write, convert to machine code and run

- write code
  - choose a good editor (or integrated development environment - IDE)
  - features like color coding, syntax checks, indexing..
  - at this point your code is just a text file
- convert to machine code
  - make sure that you have the right interpreter or compiler available
  - compilers and interpreters are binary programs
  - (for interpreted languages, this also runs the program)
- run
  - all languages run on the command line
  - some, e.g. python, run in a script mode
  - some run in IDE (integrated development env.) or in Jupyter notebooks

There will be errors

There will be lots of errors..

- **syntax** - "*Please cat dog monkey.*"
  - you broke the language rule
  - the program will not run and will return an error message

- **runtime** – "*Please eat the piano.*"
  - interpreter understands what you want but has trouble following the directions, e.g. an undefined variable or a missing module

- **semantic** – "*Please close the back door so that the bugs don't come in.*"
  - program is understood and run but produces a wrong output
  - problem is with the "meaning" of your code
  - e.g. you performed an integer instead of a float division (or you close the back door but the front one is still open)

Language building blocks

- **variables**
  variable is a name that refers to a value
  the ability to manipulate variables is one of the most powerful features of programming

- **operators**
  represent computations - “+,-,*, /,**”

- **expressions**
  combination of a variable, operator and values

```
i + 1
```

- **statements**
  unit of code that the interpreter can execute
  variable name appears alone on the left

```
i = i + 1
```

```
print(”Hello”)
```

Language building blocks continued

- **functions**
  function is a named sequence of statements
  can be repeatedly “called” from different place of the program

- **conditionals** (if, else, elif)
  let us check conditions and change the behavior of the program
  (let us control the flow of instructions)

- **loops** (for, while)
  allow for repetition of blocks of code

Variable names

- should be meaningful
- have to begin with a letter (it is a good idea to use lower case)
- can contain letters and numbers
- can contain underscores
- reserved keywords cannot be used as variable names

Python keywords: and, as, assert, break, class, continue, def, del, elif, else, except, exec (only Python 2), finally, for, from, global, if, imprort, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield, nonlocal (only Python 3)

In-built variables types

- **numeric**
  1, 3.1415, 3e-3, 1+2j
- **strings**
  ”Good morning!”
- **lists**
  [1,2,3] or [’a’, ‘b’, ‘c’] or [‘cat’, 3, [‘dog’, 2.0]]
  items in a lists can be accessed using integer indices
- **dictionaries**
  like lists but indices don’t have to be integers, list of key-value pairs (order not guaranteed)
  {‘one’:’uno’, ’two’:’dos’,’three’:’tres’}
- **tuples**
  immutable lists
  (‘a’, ‘b’, ‘c’)

Flow of instructions

Program is a sequence of instructions. Depending on the input, it is not always desirable to execute every single instruction in the same sequence.

Programming languages let us:

1. **make decisions** and take alternative paths using **conditionals**

   e.g. user inputs wrong data, the program needs to terminate.

2. **repeat** set of instructions with **loops**

   e.g. keep reading lines from a file until the end (of the file).

Conditionals

Conditional statements give us ability to check conditions and execute alternative blocks of code.

The **condition is a boolean expression** following the if keyword.

```
if x > y:
    print ("x is greater than y")
```

Conditionals

Conditional statements give us ability to check conditions and execute alternative blocks of code.

The **condition is a boolean expression** following the if keyword.

```
if x > y:
    print (”x is greater than y”)
else:
    print (”x is equal or less than y”)
```

Imperial College London

$$ax^2 + bx + c = 0$$

Case study - quadratic equation

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Depending on the value of the discriminant ($b^2 - 4ac$), quadratic equation has:

- one solution (discriminant is zero)
- two real solutions (d. is positive)
- two complex solutions (d. is negative)

Decompose the process into small steps

1. take input a, b and c
2. check input
3. compute discriminant
4. decide how many solutions
5. skip complex calculations
6. calculate real root(s)

Imperial College London

$$ax^2 + bx + c = 0$$

Example code – quadratic solver

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
import math
print(" Solve quadratic equation: ax**2 + bx + c = 0")
a = float(input(" enter a: "))
b = float(input(" enter b: "))
c = float(input(" enter c: "))

if (a == 0):    # check if imput is valid
    print("not a quadratic equation")
    exit()

d = b*b - 4*a*c  # compute discriminant

if d < 0:
    print("no real solution")
    exit()
```

Example code continued

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
# two roots
if d > 0 :
    rd = math.sqrt(d) # take the square root
    x1 = (-b + rd)/(2*a)
    x2 = (-b - rd)/(2*a)
    print("two solutions :" , x1, x2)

# one root
elif d == 0:
    x = -b/(2*a)
    print( "one solution :" , x)

# error proofing
else:
    print("something is wrong, we should not be here!")
```

Imperial College London

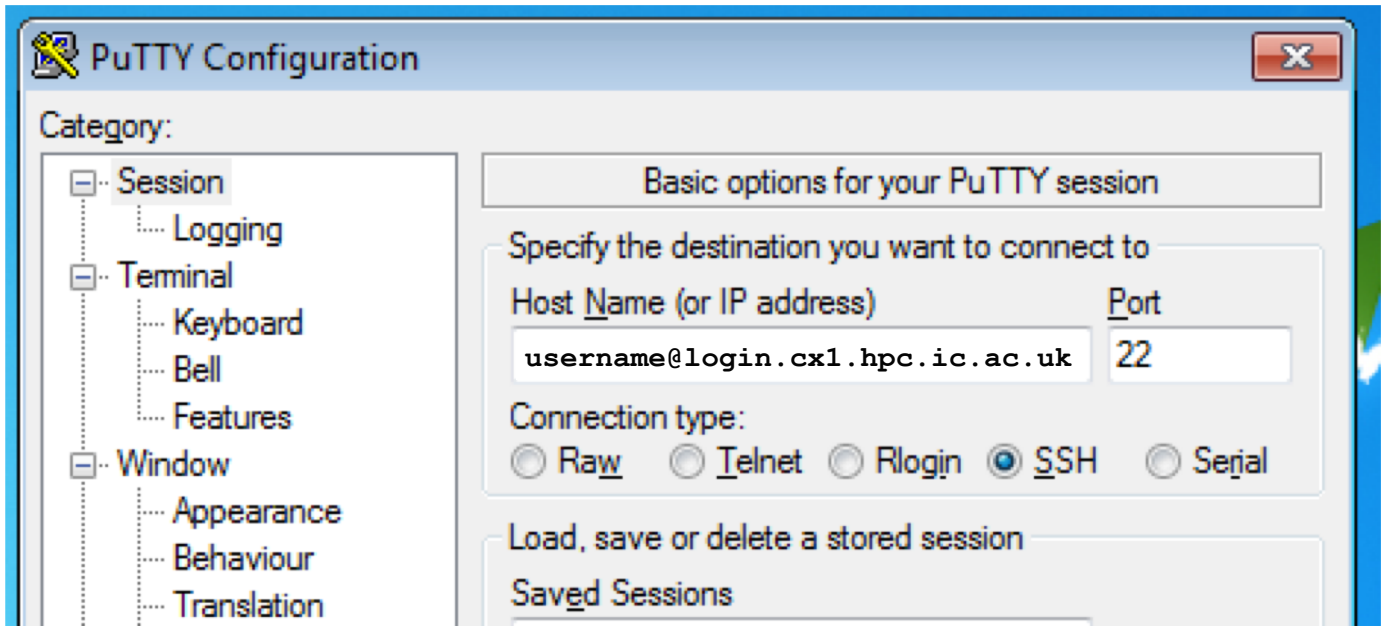
Hands-on - Log into cx1

- if on laptop, use the Imperial-WPA wifi
- we'll use the secure shell protocol (ssh) to connect to a remote system
- you'll have access for the purpose of the class

- Mac and Linux – start a terminal

```
ssh username@login.cx1.hpc.ic.ac.uk
```

- Windows - start Putty



Imperial College London

Hands-on – running scripts

1. log into cx1

2. get the code from github repository

```
git clone https://github.com/kmichali/quadratic.git
cd quadratic
```

3. load python3 environment (default is python2)

```
module load anaconda3
```

4. check version (it should be 3.5.2)

```
python --version
```

5. run quadratic.py and quadratic_short.py with different inputs

```
python quadratic.py
```

6. introduce new print statement into the code, run again (hint: `use nano quadratic.py` to edit)

In-built types

we'll look at numeric types, strings, lists, dictionaries and tuples in detail

Imperial College London

Python numerical types and operations

int – signed integers

float – real numbers

complex – in a form “a + bj“; a and b are floats and j is imaginary number

Operation	Result
`x + y`	sum of *x* and *y*
`x - y`	difference of *x* and *y*
`x * y`	product of *x* and *y*
`x / y`	quotient of *x* and *y*
`x // y`	floored quotient of *x* and *y*
`x % y`	remainder of `x / y`
`-x`	*x* negated
`+x`	*x* unchanged
`abs(x)`	absolute value or magnitude of *x*
`int(x)`	*x* converted to integer
`float(x)`	*x* converted to floating point
`complex(re, im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.

Imperial College London

Hands-on – numeric types

1. start python interactive shell (type “python”) or use Jupyter notebook

2. use python as calculator
- use each basic operator at least once
- what is floored quotient operator?
- assign a value to a variable, e.g. `a=2`
- check the type of variable using `type(a)`

Imperial College London

Hands-on – integer division

in python 3

```
>>> a=1
>>> b=2
>>> c=a/b
>>> c
0.5
>>> type(c)
<class 'float'>
>>>
```

in python 2

```
>>> a=1
>>> b=2
>>> c=a/b
>>> c
0
>>> type(c)
<type 'int'>
>>>
```

Strings

String is a sequence of characters, e.g. “Hello python” or ‘I need coffee’

You can access individual characters using an index (or rather ‘offset’ since the first element is 0).

```
str = 'python'
str[0]     # is 'p'
str[1]     # is 'y'
...
str[5]     # is 'n'

## strings can be sliced
str[0:2]   # is 'py'
str[3:]    # is 'hon'
str[-1]    # is 'n'
```

Strings continued

```
str1 = 'Python '
str2 = 'is fun'

## concatenation
str1 + str2       # 'Python is fun'

## repetition
str1 * 3          # 'Python Python Python'

## membership
'n' in str2       # True
'M' not in str2   # True
```

String methods

```
str = 'banana'

len(str)                    # 6

str.index('a'))             # 1
str.find('na')              # 2
str.replace('ba','va')      # 'vanana'
str.capitalize()            # 'Banana'
str.upper()                 # 'BANANA'
```

Very important note:
direct assignment such as
`str[0] = 'v'`
is not allowed - strings are immutable

Imperial College London

Hands-on - strings

```
>>> s = "Hello world!”          # initialise varible
>>> type(s)                     # find type
<class 'str'>
>>> len(s)                      # find length
12
>>> s[0]                        # access first element
'H'
>>> s[-1]                       # access last element
'!'
>>> s[0:5]                      # slice from start to (not included) index 5
'Hello'
>>> s[:]                        # slice the whole string
'Hello world!'
>>> s[1:]                       # slice from index 1 to the end
'ello world!'
>>> s[:5]                       # slice from start to (not included) index 5
'Hello'
>>>
```

Imperial College London

Hands-on - strings


```
>>> s = "Hello world!"
>>> s.count('l')                    # count 'l's in the string
3
>>> s.find('H')                     # find index for 'H'
0
>>> s.find('!')                     # find index for '!'
11
>>> s.split(' ')                    # split by empty spaces
['Hello', 'world!']                 # resulting data type is a list

>>> h = 'Hello'
>>> w = 'world'
>>> hw = h + ' ' + w                # concatenate strings
>>> hw
'Hello world'
```

Imperial College London

Hands-on - strings

What happens to a string when you use the replace method?

```
>>> s = "Hello world!"
>>> s.replace('Hello','Goodbye')
'Goodbye world!'
>>> s
'Hello world!'  ## s is still the same
```

To keep the new string, assign it to a new variable:

```
>>> s1 = s.replace('Hello','Goodbye')
>>> s1
'Goodbye world!'
```

Lists

List is a **sequence of values** – they can be any type (even another list) and mixed. Values in the list are called **elements** or items.

Examples:

- [1,2,3] – list of three integers
- [’mouse’, ‘lion’, ‘snake’, ‘elephant’] – list of four strings
- [‘cat’, 3, [‘dog’, 2.0]] – nested list with mixed types

Accessing element of the list:

```
animals= ['mouse', 'lion', 'snake', 'elephant']
animals[0]  # 'mouse'
animals[-1] # 'elephant'
animals[:2] # ['mouse','lion']
```

Lists are mutable

List are **mutable** - values can be changed using a direct assignment

Single element assigment:

```
animals = ['mouse', 'lion', 'snake', 'elephant']
animals[0] = 'rat'
# animals is now ['rat', 'lion', 'snake', 'elephant']
```

Slices:

```
animals[1:3] = ['pinguin','horse']
# animals is now ['rat', 'pinguin', 'horse', 'elephant']
```

Imperial College
London

List methods

```
animals = ['mouse', 'lion', 'snake', 'elephant']

animals.append('zebra’)    #append takes one argument
# ['mouse', 'lion', 'snake', 'elephant', 'zebra']

animals.pop()
# ['mouse', 'lion', 'snake', 'elephant']

animals.pop(2)
# ['mouse', 'lion', 'elephant']

animals.index('elephant')  # is 2
```

List methods

```
animals = ['mouse', 'lion', 'elephant']
animals.extend(['pinguin','whale','mole'])
#  ['mouse', 'lion', 'elephant', 'pinguin', 'whale', 'mole']

animals.remove('whale')
#  ['mouse', 'lion', 'elephant', 'pinguin', 'mole']

animals.insert(1,'tiger')
# ['mouse', 'tiger', 'lion', 'elephant', 'pinguin', 'mole']

animals.sort()
# ['elephant', 'lion', 'mole', 'mouse', 'pinguin' 'tiger']

animals.reverse()
# ['tiger', 'pinguin', 'mouse', 'mole', 'lion', 'elephant']
```

Concatenation and repetition

```
['monkey'] + ['human','ape']
#['monkey','human','ape']


['bat', 'cat'] * 2
# ['bat', 'cat', 'bat', 'cat']
```

List functions

```
l = list(range(1,11))      # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Length of a list:

```
len(l)                     # returns 10
```

Maximum:

```
max(l)                     # 10
```

Minimum:

```
min(l)                     # 1
```

For min and max to work, the list has to contain the items of the same type.

Imperial College London

Hands-on – lists exercise

1. Instantiate a list that contains strings representing colors (don't forget to include 'blue')

2. Find index for 'blue'

3. Replace 'blue' with 'green'

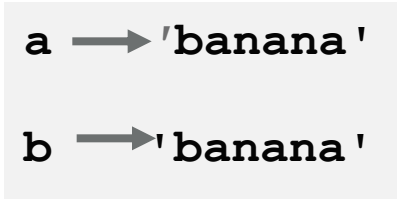
4. If the list was reversed, what index would 'green' occupy? Calculate the answer ahead of time using python (hint: use length function).

5. Reverse the list and check your answer

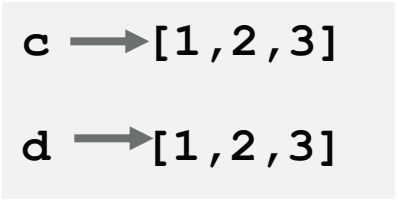
Imperial College London

Objects and values – what happens in the memory?

```
# strings
a = 'banana'
b = 'banana'
```




```
# lists
c = [1,2,3]
d = [1,2,3]
```



Imperial College London

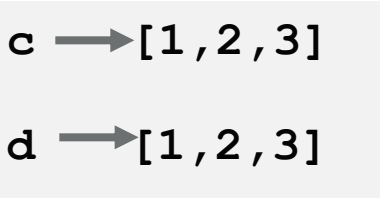
Objects and values

```
# strings
a = 'banana'
b = 'banana'

a is b
True  #the same object
```


```
# lists
c = [1,2,3]
d = [1,2,3]

c is d
False  #two different objects
```



Strings are immutanble, lists are mutable

```
# strings
a = 'banana'
```

a → 'banana'

```
# this produces a new object
b = a.capitalize()
```

a → 'banana'

b → 'Banana'

```
# if you apply a method on a list
# it modifies the original list
c = [1,2,3]
c.append(4)
# note: don't use with assignment
#c.append doesn't return a list
```

c → [1,2,3]

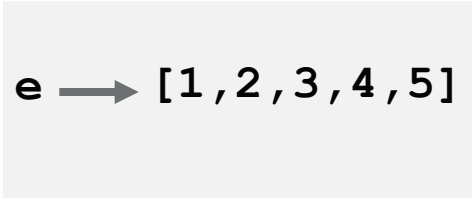
c → [1,2,3,4]

Imperial College London

Aliasing

```
e = [1,2,3,4]
f = e
e is f
True
```


```
e.append(5)
e
[1,2,3,4,5]
# what happened to f?
```

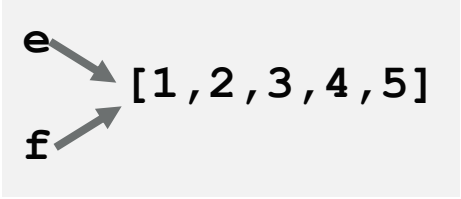


Imperial College London

Aliasing

```
e = [1,2,3,4]
f = e
e is f
True
```


```
e.append(5)
e
[1,2,3,4,5]
# what happened to f?
f
[1,2,3,4,5]
```



Imperial College London

Dictionaries

Dictionaries are like lists but indices don't have to be integers.

They consist of key-value pairs. Each key maps to a value.

Example: {'one':'uno', 'two':'dos','three':'tres'}

Dictionaries are one Python's best feature; they are building blocks of many efficient and elegant algorithms.

Initiate a dictionary

Create a new dictionary:

```
eng2sp = dict()
```

Add an item:

```
eng2sp['one'] = 'uno'
```

Replace an item:

```
eng2sp['one'] = 'ein'
```

Add many items at once:

```
eng2sp = {'two':'dos', 'three':'tres'}
```

Delete an item:

```
del eng2sp['two']
```

Dictionary keys and dictionary implementation

- No key duplicates are allowed, the last assignment wins
- Keys must be immutable (numbers, strings, tuples) because they are used to produce hash values that cannot change
- Hash values are placed into sorted array
- This makes searching for dictionary keys very efficient. The time to look up a key stays constant when a dictionary grows bigger

Dictionary functions and methods

Number of items in a dictionary:

```
len(eng2sp)
```

Remove all elements:

```
eng2sp.clean()
```

Return a shallow copy:

```
eng2sp.copy()
```

Return complete copy:

```
eng2sp.deepcopy()
```

Return value by key (if not found, return default):

```
eng2sp.get(key, default=None)
```

Dictionary methods

Return true is key found, false otherwise:

```
eng2sp.has_key(key)
```

Return list of keys:

```
eng2sp.keys()
```

Return list of values:

```
eng2sp.values()
```

Return list of tuples holding key-value pairs:

```
eng2sp.items()
```

Imperial College London

Hands-on - dictionaries

1. instantiate a dictionary (if you cannot think of anything, use translation from English to your favorite language)
2. examine the dictionary using key and values methods
3. add a pair that has the same key as an existing element
4. what happened to the dictionary?
5. add a pair that has the same value as an existing element
6. examine the dictionary again

Tuples

Tuples are similar to lists. They are sequences of values of any type and are indexed by integers. The difference is that values in tuples cannot be changed. They are immutable.

Create an empty tuple:

```
t = tuple()
```

Tuple assignment (parentheses can be skipped):

```
t = ('a', 'b', 'c', 'd')
```

Tuple with a single element:

```
t1 = ('z',)
```

Without the final comma, t1 will be a string

Tuples

from string to tuple:

```
t2 = tuple('python')
# t2 is now ('p', 'y', 't', 'h', 'o', 'n')
```

Accessing elements:

```
t2[0]           # 'p'
t2[1]           # 'y'
t2[5]           # 'n'
```

Slicing:

```
t2[1:3]           # ('y','t')
t2[-2:-1]         # ('o',)
```

Tuple tricks

Swapping variables:
With conventional assignment, a temporary variable is needed:

```
temp = a
a = b
b = temp
```

Using tuples:

```
(a, b) = (b, a)
```

Quick assignment:

```
addr = 'monty@python.org'
(uname, domain) = addr.split('@')
# uname is monty and domain is python.org
```

Why tuples?

- useful as function return values
- if one needs to use a sequence (list or tuple) as a dictionary key, it has to be tuple
- if one needs a sequences as function argument, tuples are safer

Imperial College London

Hands-on - tuples

You have four variables a, b, c, d. Switch the values so that:
a = d , b = c, c = b, d = a

Do this first without tuples and then with tuples.

Flow of instructions

Program is a sequence of instructions. Depending on the input, it is not always desirable to execute every single instruction in the same sequence.

Programming languages let us:

1. **make decisions** and take alternative paths using **conditionals**
   e.g. user inputs wrong data, the program needs to terminate.

2. **repeat** set of instructions with **loops**
   e.g. keep reading lines from a file until the end (of the file).

For loop

For loop is a block of statements repeated a pre-determined number of times.

```
for i in range(10):
    print(i)
# will produce numbers from 0 to 9 (one per line)
```

```
for i in range(1,11):
    print(i)
# will produce numbers from 1 to 10 (one per line)
```

For loops are often used to iterate through lists and strings.

String traversal

```
str = 'python'
for char in str:
    print(char)

# outputs
p
y
t
h
o
n
```

Travesing list

```
mix = ['cat', 3, ['dog', 2.0]]

for i in mix:
    print (i)

# outputs
cat
3
['dog', 2.0]
```

But what about the nested list?

Traversing nested list (naïve method..)

```
mix = ['cat', 3, ['dog', 2.0]]
for i in mix:
    print(i)
    if type(i) is list:
        for j in i:
            print(j)

#outputs
cat
3
['dog', 2.0]
dog
2.0
```

While loop

While loop is controlled by a boolean expression, it iterates until the expression becomes false. The expression is checked on every iteration.

```
i = 0
while i < 11:
    print(i)
    i = i + 1
# will print numbers from 0 to 10 (one per line)

while True:
    reply = input('Enter text, [type "stop" to quit]:')
    print(reply.lower())
    if reply == 'stop':
        break
# will ask for new text until you type ”stop”
```

Imperial College London

Hands-on - loops

1. look up the range function and write a loop that iterates through numbers 1000, 990, 980, 970, …, 0

2. explore the time cost of loops:
   - write loop that adds numbers from 1 to 10^6 and time it
   - use Gaussian addition sum = ½ (n*(n+1)) and compare the timings

to time execution:

```
import time
start_time = time.time()
## your code
print("--- %s seconds ---" % (time.time() - start_time))
```

3. write an infinite loop that prints numbers incremented by one, let it run for a while, force stop (in Notebook you can stop the kernel and restart it, otherwise use ^C) and reflect on the fact that the loop would have run until hardware ceased

Functions

Sequence of statements that performs a computation. A function can be called by name.

Python already includes many in-built functions:

- print() - prints to the screen
- type() - for testing variable type, e.g. type(10) returns ‘int’
- int() - converts to integer (if it is possible), e.g. int(‘22’) converts string ‘22’ to integer

Or can write your own functions.

Python comes with myriad of packages with functions. Always check if you can use an existing code before writing your own.

Write your own functions

- grouping statements makes the program easier to read, test and debug
- functions eliminate repetitive code
- well-designed functions can be shared by many programs
- function definition should appear before the function is called

```
def add_nums(a,b):   # define function
    return a + b     # return value
                     # indentation (and function) ends here

sum = add_nums(3,4) # call the function, sum is now 7
```

```
def print_stuff(name, greeting = "Hello"):
    message = greeting + " " + name + "!"
    print(message)

print_stuff("Bob")                  # outputs "Hello Bob!"
print_stuff("Bob", "Adios")         # outputs "Adios Bob!"
```

Imperial College London

$$ax^2 + bx + c = 0$$

Example code – quadratic solver

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
import math
print(" Solve quadratic equation: ax**2 + bx + c = 0")
a = float(input(" enter a: "))
b = float(input(" enter b: "))
c = float(input(" enter c: "))

if (a == 0):    # check if imput is valid
    print("not a quadratic equation")
    exit()

d = b*b - 4*a*c  # compute discriminant

if d < 0:
    print("no real solution")
    exit()
```

Imperial College London

Example code – no functions

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
# two roots
if d > 0 :
    rd = math.sqrt(d) # take the square root
    x1 = (-b + rd)/(2*a)
    x2 = (-b - rd)/(2*a)
    print("two solutions :" , x1, x2)
# one root
elif d == 0:
    x = -b/(2*a)
    print( "one solution :" , x)
# error proofing
else:
    print("something is wrong, we should not be here!")
```

Imperial College London

Example code – with functions

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
def two_soln(aa,bb,dd):
    rr = math.sqrt(dd) # take the square root
    x1 = (-bb + rr)/(2*aa)
    x2 = (-bb - rr)/(2*aa)
    return x1,x2

def one_soln(aa,bb):
    x = -bb/(2*aa)
    return x

if d > 0 :
    print("two solutions :" , two_soln(a,b,d))
elif d == 0:
    print( "one solution :" , one_soln(a,b))
else:
    print("something is wrong, we should not be here!")
```

Imperial College London

Hands-on - functions

1. write a program that uses one function that takes a number and squares it

2. improve the function so it will raise a number to a power of N (where N is an integer)

Flow of instructions

Program is a sequence of instructions. Depending on the input, it is not always desirable to execute every single instruction in the same sequence.

Programming languages let us:

1. **make decisions** and take alternative paths using **conditionals**
   e.g. user inputs wrong data, the program needs to terminate.

2. **repeat** set of instructions with **loops**
   e.g. keep reading lines from a file until the end (of the file).

Conditionals

Conditional statements give us ability to check conditions and execute alternative blocks of code.

The **condition is a boolean expression** following the if keyword.

```
if x > y:
    print ("x is greater than y")
```

Conditionals

Conditional statements give us ability to check conditions and execute alternative blocks of code.

The **condition is a boolean expression** following the if keyword.

```
if x > y:
    print (”x is greater than y”)
else:
    print (”x is equal or less than y”)
```

Conditionals

Conditional statements give us ability to check conditions and execute alternative blocks of code.

The **condition is a boolean expression** following the if keyword.

```
if x > y:
    print (”x is greater than y”)
elif x < y:
    print (”x is less than y”)
else:
    print (”x is equal to y”)
```

Boolean expression

Boolean expression is either true or false

9 == 9 is True

8 == 9 is False

True and False are special values of type “bool”.

Boolean operators:

- == equal (note **double** equal sign)
- > greater than
- < less than
- >= greater than or equal to
- <= less than or equal to

You can combine boolean expressions with **logical operators**: and, or, not

x = 6

x < 10 and x > 6 is False x < 5 or x >= 6 is True not (x == 6) is False

Imperial College London

Hands-on - conditionals

Write program that calculates the value of stamp duty on a house.
Input: price
Output: stamp duty

Start with developing a program for houses that are cheaper than £250,000.

Property or lease premium or transfer value	SDLT rate
Up to £125,000	Zero
The next £125,000 (the portion from £125,001 to £250,000)	2%
The next £675,000 (the portion from £250,001 to £925,000)	5%
The next £575,000 (the portion from £925,001 to £1.5 million)	10%
The remaining amount (the portion above £1.5 million)	12%

File I/O

```
file = open("testfile.txt","w")
file.write("Hello World")
file.close()

filein = open("testfile.text", "r")
print filein.read()
filein.close()
```

Modes:

- w – write only (overwrites existing file)
- r – read only
- a – append
- r+ – read and write

Imperial College London

Importing packages

Use existing code, don't reinvent. It is likely that your code will not be as “robust”.

```
import math
help('math')
print(math.pi)
print(math.sin(1.2))   ##to access math functions, use dot notation


from math import *
print(pi)
print(sin(1.2))   ## no need for dot notation but generally discouraged


import numpy as np   ## shorten name, use as np
print(np.pi)
```

Imperial College London

Parameter checking with raise and assert

```
def myFunction(filepath):
    infile = open(filepath)
```

What if filepath does not exist?

```
## check with raise
def myFunction(filepath):
  if not os.path.exists(filepath):
    raise IOError('File does not exist')
```

```
## check with assert
def myFunction(filepath):
  assert os.path.exists(filepath), 'File does not exist'
```

Parameter checking with try + except

```
## check with try + except
def myFunction(filepath):
  try:
    with open(filepath) as infile:
      pass
  except IOError:
    print('File does not exist')
```

Alternative code for quadratic solver – concise but not user friendly and run-time error prone

```
import cmath
a = float(input('Enter a: '))
b = float(input('Enter b: '))
c = float(input('Enter c: '))

# calculate the discriminant
d = (b**2) - (4*a*c)

# find two solutions
sol1 = (-b-cmath.sqrt(d))/(2*a)
sol2 = (-b+cmath.sqrt(d))/(2*a)

print('The solution are {0} and {1}'.format(sol1,sol2))
```

Hands-on - programming style

1. add error checking to the quadratic solver on previous slide
2. make the code user-friendly with print
3. add comments to explain what different parts of the program are doing

What next?

After learning the fundamentals of the language (plus a bit of practice), one can proceed to learn specialised packages depending on their field of interest.

- Matplotlib – Matlab style plotting
- NumPy – implementation of arrays and matrices
- SciPy – library for scientific computing (optimisation, linear algebra, integration, interpolation, special fuctions, FFT, signal and image processing, ODE solvers..)
- Pandas – data manipulation and analysis
- SciKit-Learn, TensorFlow – machine and deep learning

Resources

Think Python by Allen B. Downey
http://greenteapress.com/wp/think-python-2e/

Python for Informatics: Exploring Information by Charles Severance
https://www.pythonlearn.com/html-009/index.html
(inspired by Think Python)

Software carpentry python materials
https://swcarpentry.github.io/python-novice-inflammation/

Imperial College London

Resources

Numpy workbook

https://nbviewer.jupyter.org/urls/imperialcollegelondon.box.com/shared/static/uqobo1kbfvucqfqy2bnnn5p46c60qeqo.ipynb

Matplotlib workbook

https://nbviewer.jupyter.org/urls/imperialcollegelondon.box.com/shared/static/h6roa8ovmtg8ktk6dkzvm2qapyr4ctfy.ipynb

Pandas workbook

https://nbviewer.jupyter.org/urls/imperialcollegelondon.box.com/shared/static/68058kq8m7jz4lhyo64y4gapvr6krkr9.ipynb