Introduction to

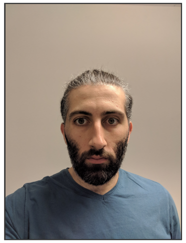
python

with Application to Bioinformatics

- Day 1

Who we are

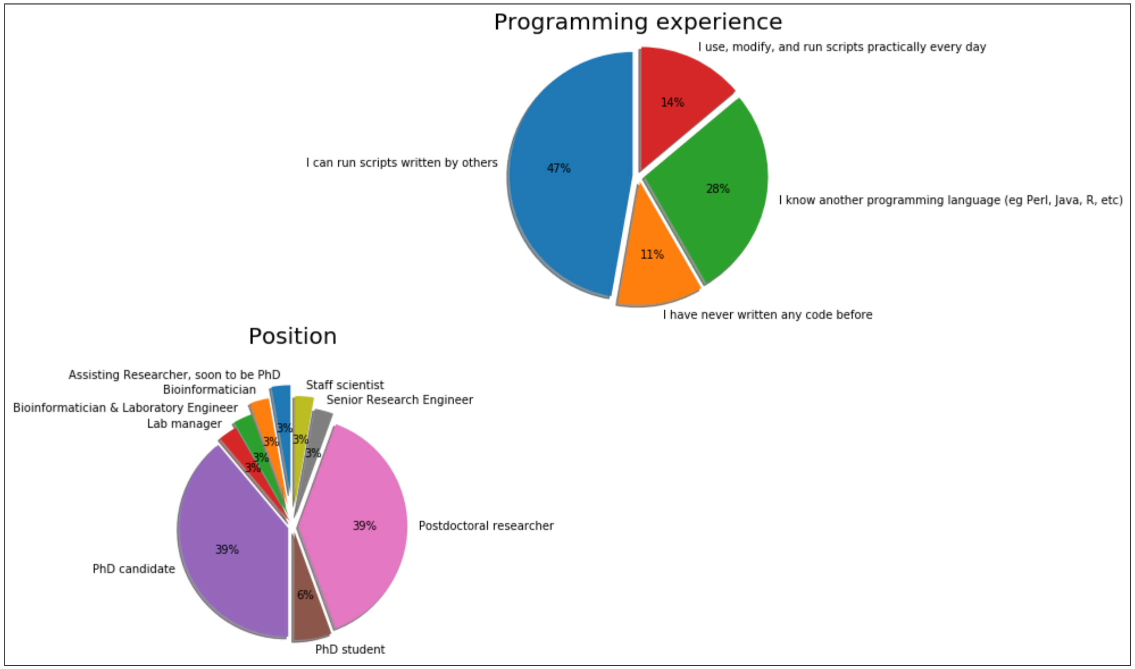
Uppsala

Nina	Malin	Verena	Dimitrios	Johan	Martin
					—

Umeå

Jeanette	Allison	Matus	Pedro

Who you are


Programming experience
I use, modify, and run scripts practically every day
14%
I can run scripts written by others
47%
28%
I know another programming language (eg Perl, Java, R, etc)
11%
I have never written any code before
Position
Assisting Researcher, soon to be PhD
Bioinformatician
Bioinformatician & Laboratory Engineer
Lab manager
Staff scientist
Senior Research Engineer
3%
3%
3%
3%
3%
3%
3%
39%
Postdoctoral researcher
39%
PhD candidate
6%
PhD student

Schedule

	Monday	Tuesday	Wednesday	Thursday	Friday
09.00 - 12.00	🟩 🟥 🟨 🟩 🟥 🟨 🟩 🟥 🟨		Lectures with Hands-on Exercises		
12.00 - 13.00			LUNCH		
13.00 - 15.00	🟩 🟥 🟨 🟩 🟥 🟨 🟩 🟥 🟨		Lectures with Hands-on Exercises		
15.00 - 17.00			Own Practice on Main Assignment		
	🟩 Lecture	🟥 Example (me)	🟨 Practice (you)		

Check

- Has everyone managed to install Python?
- Have you managed to run the test script?
- Have you installed notebooks? (optional)

What is programming?

Wikipedia:

"Computer programming is the process of building and designing an executable computer program for accomplishing a specific computing task"

What can we use it for?

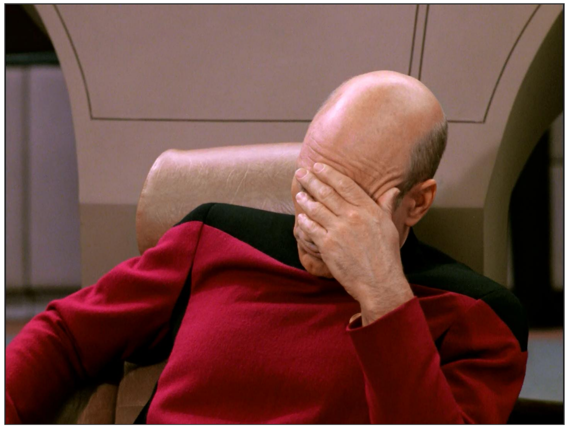
Endless possibilities!

- reverse complement DNA
- custom filtering of VCF files
- plotting of results
- all excel stuff!

Why Python?

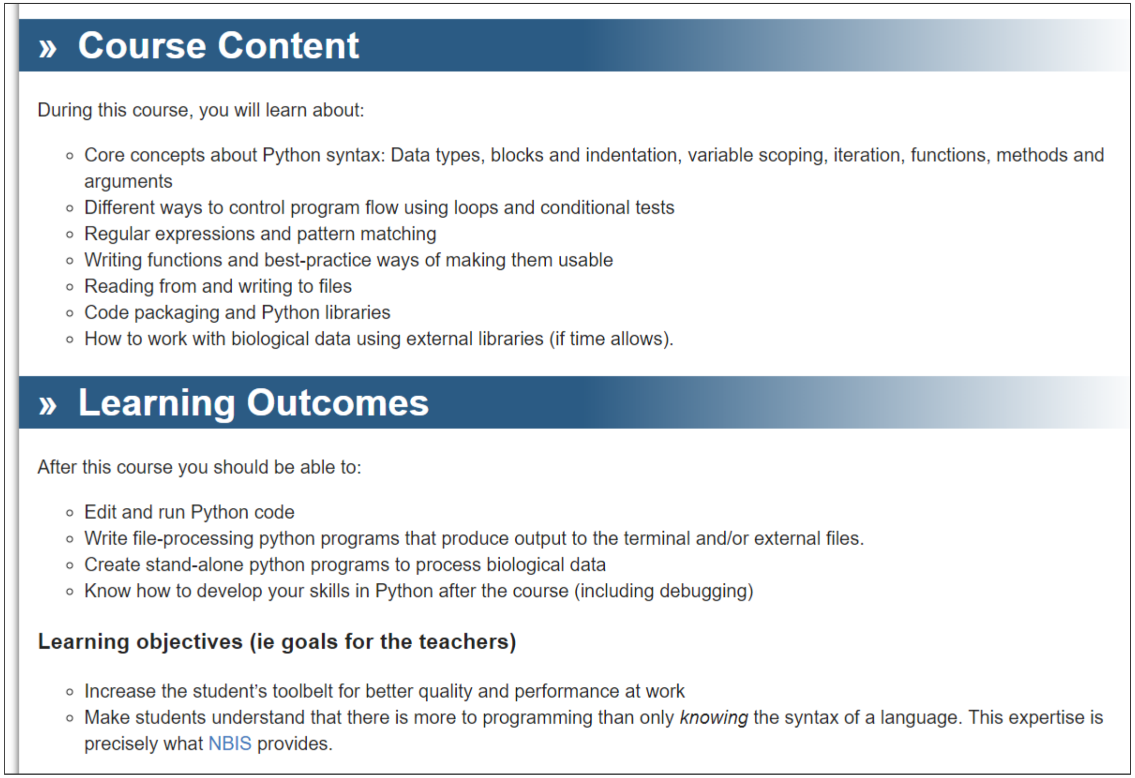
Typical workflow

1. Get data
2. Clean, transform data in spreadsheet
3. Copy-paste, copy-paste, copy-paste
4. Run analysis & export results
5. Realise the columns were not sorted correctly
6. Go back to step 2, Repeat



Python versions

- Python 1.0 - January 1994
- Python 1.2 - April 10, 1995
- Python 1.3 - October 12, 1995
- Python 1.4 - October 25, 1996
- Python 1.5 - December 31, 1997
- Python 1.6 - September 5, 2000
- Python 2.0 - October 16, 2000
- Python 2.1 - April 17, 2001
- Python 2.2 - December 21, 2001
- Python 2.3 - July 29, 2003
- Python 2.4 - November 30, 2004
- Python 2.5 - September 19, 2006
- Python 2.6 - October 1, 2008
- Python 2.7 - July 3, 2010
- Python 3.0 - December 3, 2008
- Python 3.1 - June 27, 2009
- Python 3.2 - February 20, 2011
- Python 3.3 - September 29, 2012
- Python 3.4 - March 16, 2014
- Python 3.5 - September 13, 2015
- Python 3.6 - December 23, 2016
- Python 3.7 - June 27, 2018



» Course Content

During this course, you will learn about:

- Core concepts about Python syntax: Data types, blocks and indentation, variable scoping, iteration, functions, methods and arguments
- Different ways to control program flow using loops and conditional tests
- Regular expressions and pattern matching
- Writing functions and best-practice ways of making them usable
- Reading from and writing to files
- Code packaging and Python libraries
- How to work with biological data using external libraries (if time allows).

» Learning Outcomes

After this course you should be able to:

- Edit and run Python code
- Write file-processing python programs that produce output to the terminal and/or external files.
- Create stand-alone python programs to process biological data
- Know how to develop your skills in Python after the course (including debugging)

Learning objectives (ie goals for the teachers)

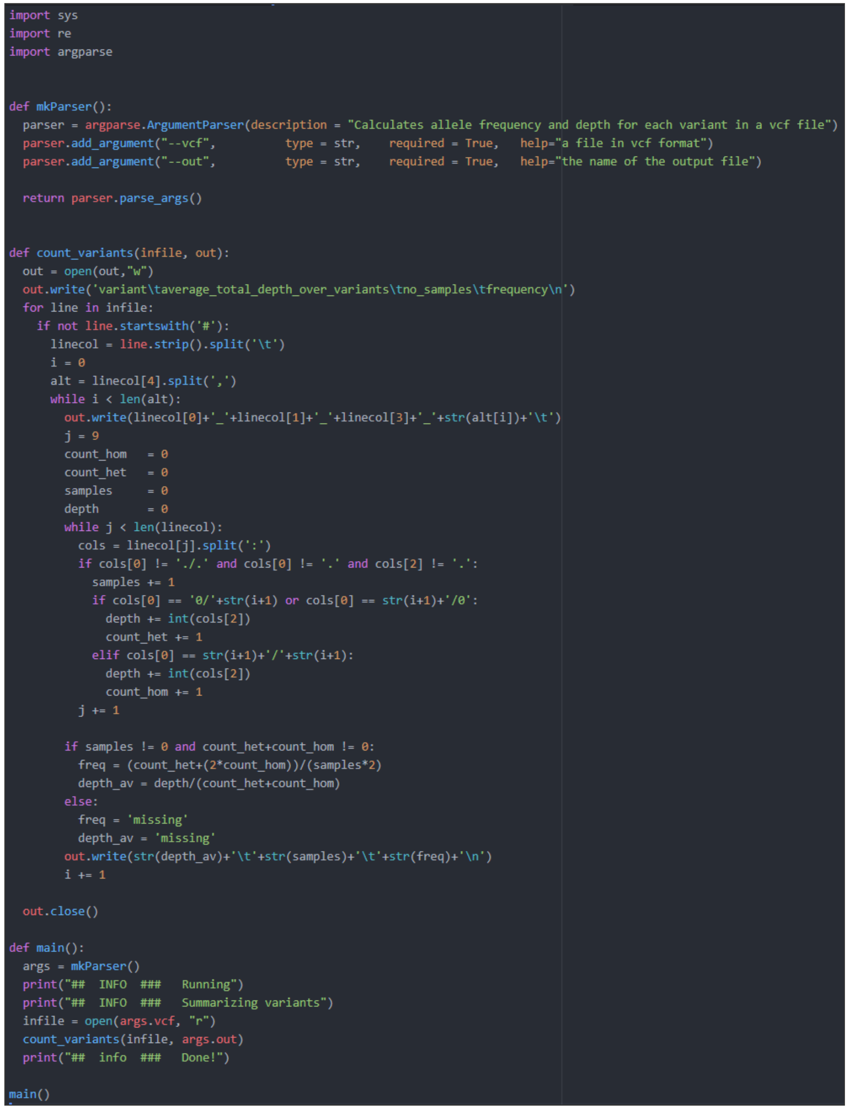
- Increase the student's toolbelt for better quality and performance at work
- Make students understand that there is more to programming than only *knowing* the syntax of a language. This expertise is precisely what NBIS provides.

Some good advice

- 5 days to learn Python is not much
- Amount of information will decrease over days
- Complexity of tasks will increase over days
- Read the error messages!
- Save all your code

How to seek help:

- Google
- Ask your neighbour
- Ask an assistant



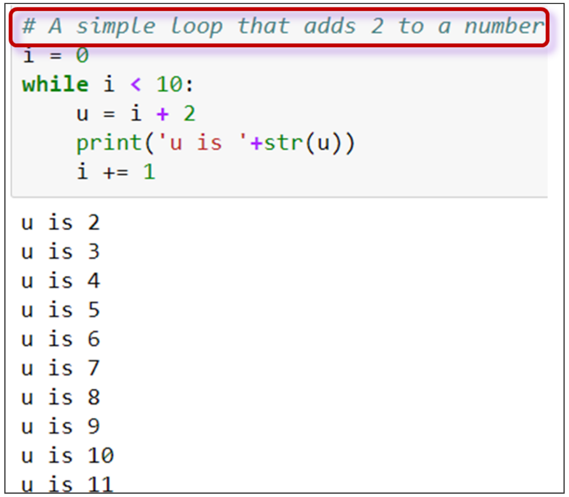
```
import sys
import re
import argparse


def mkParser():
  parser = argparse.ArgumentParser(description = "Calculates allele frequency and depth for each variant in a vcf file")
  parser.add_argument("--vcf",          type = str,    required = True,   help="a file in vcf format")
  parser.add_argument("--out",          type = str,    required = True,   help="the name of the output file")

  return parser.parse_args()


def count_variants(infile, out):
  out = open(out,"w")
  out.write('variant\taverage_total_depth_over_variants\tno_samples\tfrequency\n')
  for line in infile:
    if not line.startswith('#'):
      linecol = line.strip().split('\t')
      i = 0
      alt = linecol[4].split(',')
      while i < len(alt):
        out.write(linecol[0]+'_'+linecol[1]+'_'+linecol[3]+'_'+str(alt[i])+'\t')
        j = 9
        count_hom   = 0
        count_het   = 0
        samples     = 0
        depth       = 0
        while j < len(linecol):
          cols = linecol[j].split(':')
          if cols[0] != './.' and cols[0] != '.' and cols[2] != '.':
            samples += 1
            if cols[0] == '0/'+str(i+1) or cols[0] == str(i+1)+'/0':
              depth += int(cols[2])
              count_het += 1
            elif cols[0] == str(i+1)+'/'+str(i+1):
              depth += int(cols[2])
              count_hom += 1
          j += 1

        if samples != 0 and count_het+count_hom != 0:
          freq = (count_het+(2*count_hom))/(samples*2)
          depth_av = depth/(count_het+count_hom)
        else:
          freq = 'missing'
          depth_av = 'missing'
        out.write(str(depth_av)+'\t'+str(samples)+'\t'+str(freq)+'\n')
        i += 1

  out.close()

def main():
  args = mkParser()
  print("##  INFO  ###   Running")
  print("##  INFO  ###   Summarizing variants")
  infile = open(args.vcf, "r")
  count_variants(infile, args.out)
  print("##  info  ###   Done!")

main()
```

Example of a simple Python script

In [ ]:

```
# A simple loop that adds 2 to a number
i = 0
while i < 10:
    u = i + 2
    print('u is',u)
    i += 1
```

Example of a simple Python script



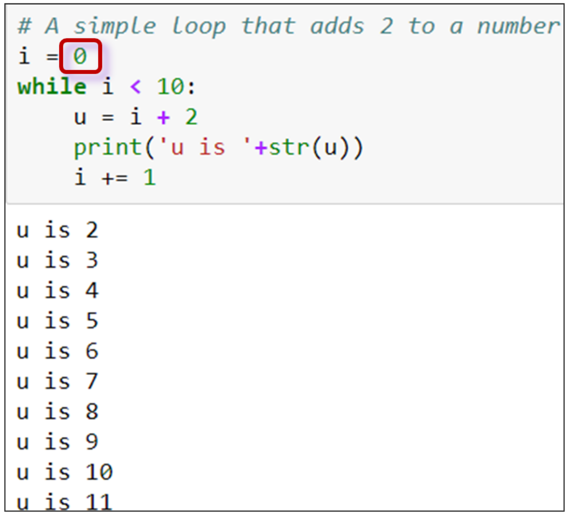
```
# A simple loop that adds 2 to a number
i = 0
while i < 10:
    u = i + 2
    print('u is '+str(u))
    i += 1
```

```
u is 2
u is 3
u is 4
u is 5
u is 6
u is 7
u is 8
u is 9
u is 10
u is 11
```

Comment

All lines starting with # is interpreted by python as a comment and are not executed. Comments are important for documenting code and considered good practise when doing all types of programming

Example of a simple Python script



```python
# A simple loop that adds 2 to a number
i = 0
while i < 10:
    u = i + 2
    print('u is '+str(u))
    i += 1
```

```
u is 2
u is 3
u is 4
u is 5
u is 6
u is 7
u is 8
u is 9
u is 10
u is 11
```

Literals

All literals have a type:

Strings (str)	‘Hello’ “Hi”
Integers (int)	5
Floats (float)	3.14
Boolean (bool)	True or False

Literals define values

In [ ]:

```
'this is a string'
"this is also a string"
3       # here we can put a comment so we know that this is an integer
3.14    # this is a float
True    # this is a boolean
```

Collections

In [ ]:

```
[3, 5, 7, 4, 99]       # this is a list of integers

('a', 'b', 'c', 'd')   # this is a tuple of strings
{'a', 'b', 'c'}        # this is a set of strings
{'a':3, 'b':5, 'c':7}  # this is a dictionary with strings as keys and integers as values
```

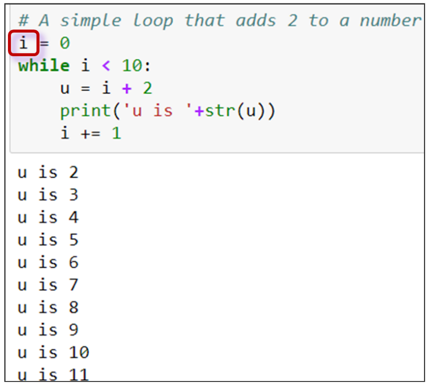
What operations can we do with different values?

That depends on their type:

```
In [ ]: 'a string'+' another string'
```

Type	**Operations**
int	+ - / ** % // ...
float	+ - / * % // ...
string	+

Example of a simple Python script



```
# A simple loop that adds 2 to a number
i = 0
while i < 10:
    u = i + 2
    print('u is '+str(u))
    i += 1
```

```
u is 2
u is 3
u is 4
u is 5
u is 6
u is 7
u is 8
u is 9
u is 10
u is 11
```

Identifiers

Identifiers are used to identify a program element in the code.

For example:

- Variables
- Functions
- Modules
- Classes

Variables

Used to store values and to assign them a name.

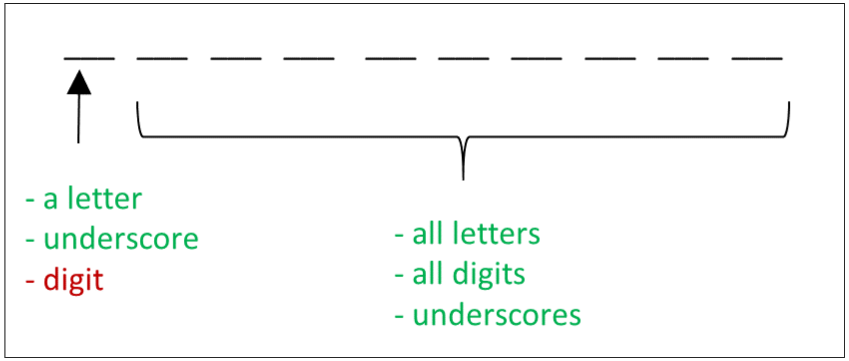
Examples:

- `i       = 0`
- `counter = 5`
- `snpname = 'rs2315487'`
- `snplist = ['rs21354', 'rs214569']`

In [ ]:

```
width  = 23564
height = 10

snpname = 'rs56483'
snplist = ['rs12345','rs458782']
```

How to correctly name a variable



- a letter
- underscore
- digit

- all letters
- all digits
- underscores

Allowed:

Var_name
_total
aReallyLongName
with_digit_2
dkfsjdsklut *(well, allowed, but NOT recommended)*

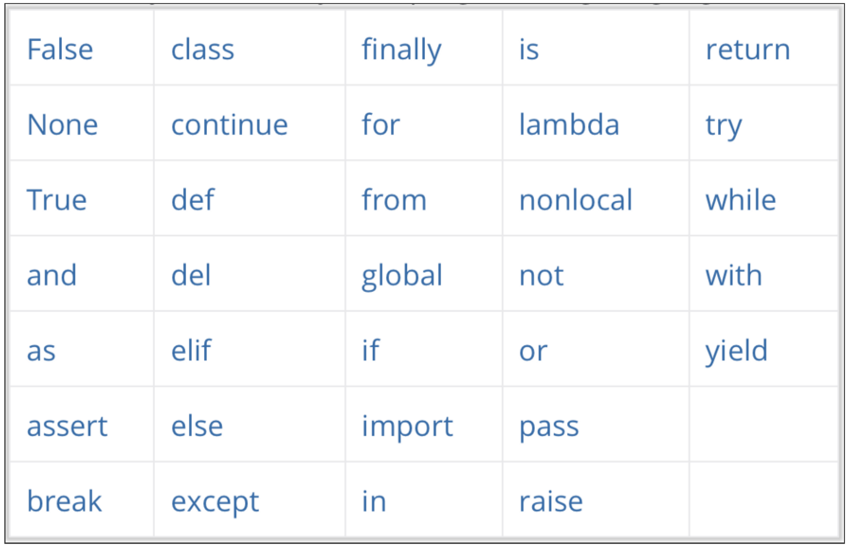
Not allowed:

2save
*important
Special%
With spaces

NO special characters:

+ - * $ % ; : , ? ! { } ( ) < > “ ‘ | \ @

Reserved keywords



False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

These words can not be used as variable names

Summary

- Comment your code!
- Literals define values and can have different types (strings, integers, floats, boolean)
- Values can be collected in lists, tuples, sets, and dictionaries
- The operation that can be performed on a certain value depends on the type
- Variables are identified by a name and are used to store a value or collections of values
- Name your variables using descriptive words without special characters and reserved keywords

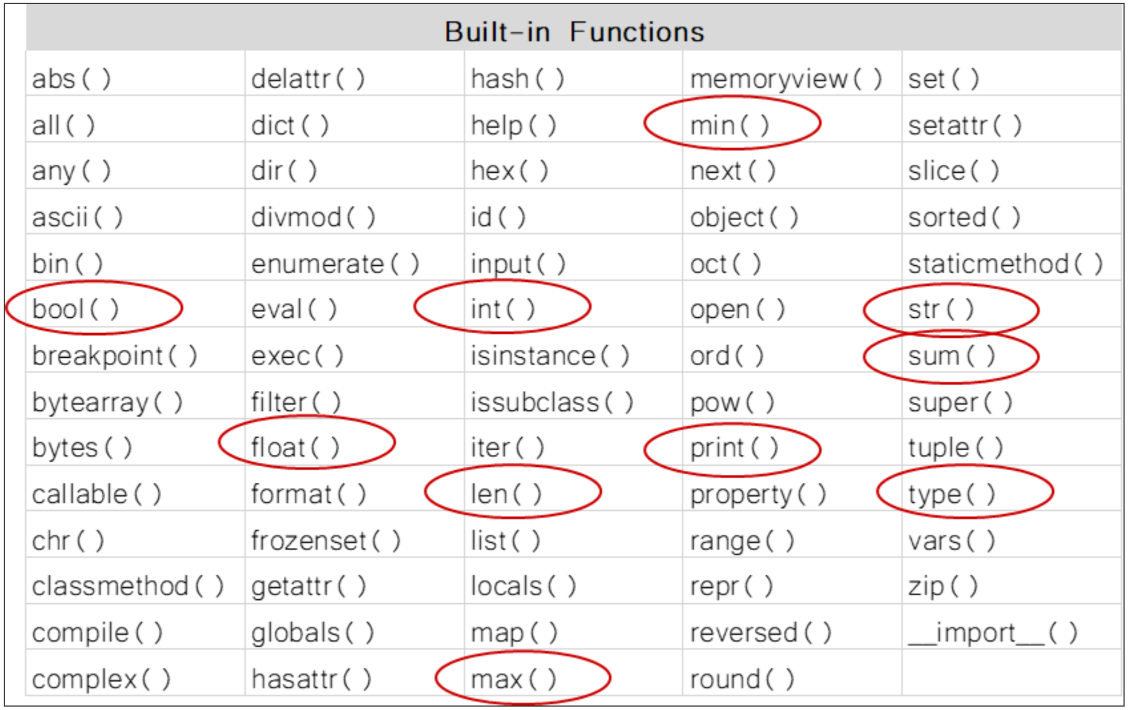
→ **Notebook Day_1_Exercise_1 (~30 minutes)**

NOTE!

How to get help?

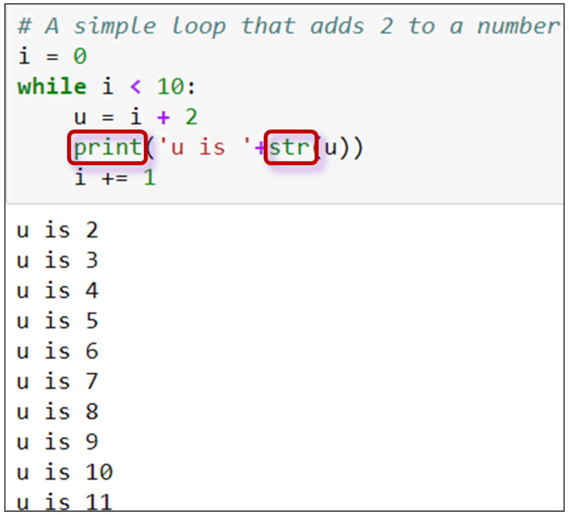
- Google (https://www.google.com/) and Stack overflow (https://stackoverflow.com/) are your best friends!
- Official python documentation (https://docs.python.org/3/)
- Ask your neighbour
- Ask us

Python standard library



Built-in Functions				
abs( )	delattr( )	hash( )	memoryview( )	set( )
all( )	dict( )	help( )	[✓ min( )]	setattr( )
any( )	dir( )	hex( )	next( )	slice( )
ascii( )	divmod( )	id( )	object( )	sorted( )
bin( )	enumerate( )	input( )	oct( )	staticmethod( )
[✓ bool( )]	eval( )	[✓ int( )]	open( )	[✓ str( )]
breakpoint( )	exec( )	isinstance( )	ord( )	[✓ sum( )]
bytearray( )	filter( )	issubclass( )	pow( )	super( )
bytes( )	[✓ float( )]	iter( )	[✓ print( )]	tuple( )
callable( )	format( )	[✓ len( )]	property( )	[✓ type( )]
chr( )	frozenset( )	list( )	range( )	vars( )
classmethod( )	getattr( )	locals( )	repr( )	zip( )
compile( )	globals( )	map( )	reversed( )	__import__( )
complex( )	hasattr( )	[✓ max( )]	round( )	

Example `print()` and `str()`

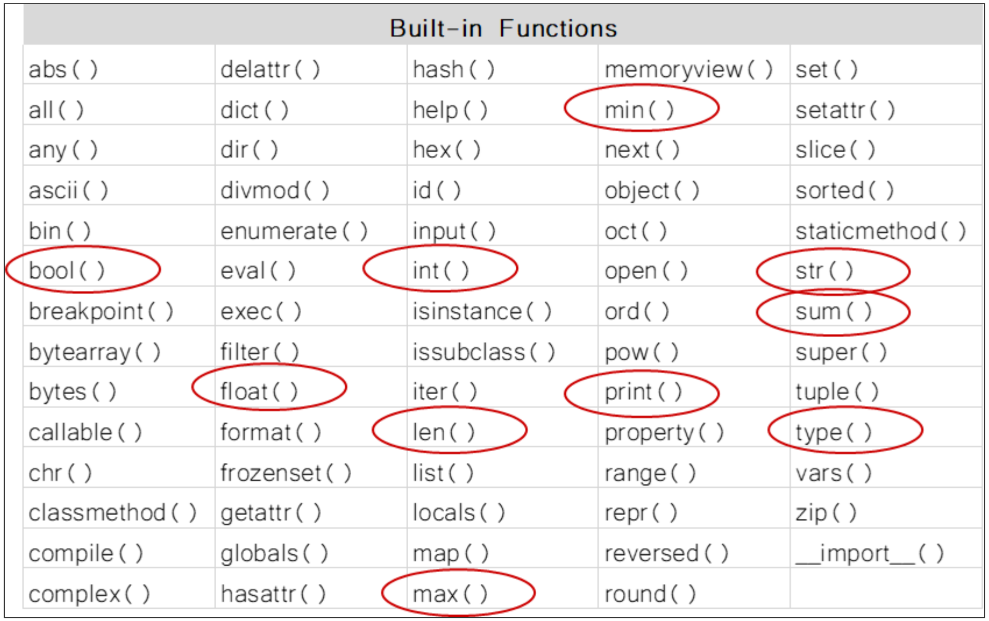


```
# A simple loop that adds 2 to a number
i = 0
while i < 10:
    u = i + 2
    print('u is '+str(u))
    i += 1
```

```
u is 2
u is 3
u is 4
u is 5
u is 6
u is 7
u is 8
u is 9
u is 10
u is 11
```

Note!
Here we format everything to a string before printing it

Python standard library

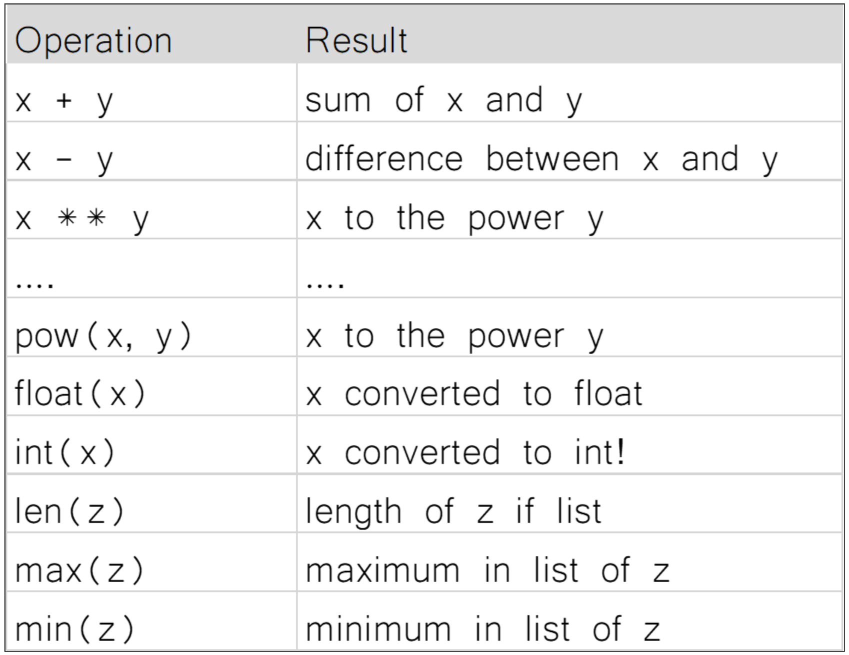


Built-in Functions				
abs( )	delattr( )	hash( )	memoryview( )	set( )
all( )	dict( )	help( )	[✓ min( )]	setattr( )
any( )	dir( )	hex( )	next( )	slice( )
ascii( )	divmod( )	id( )	object( )	sorted( )
bin( )	enumerate( )	input( )	oct( )	staticmethod( )
[✓ bool( )]	eval( )	[✓ int( )]	open( )	[✓ str( )]
breakpoint( )	exec( )	isinstance( )	ord( )	[✓ sum( )]
bytearray( )	filter( )	issubclass( )	pow( )	super( )
bytes( )	[✓ float( )]	iter( )	[✓ print( )]	tuple( )
callable( )	format( )	[✓ len( )]	property( )	[✓ type( )]
chr( )	frozenset( )	list( )	range( )	vars( )
classmethod( )	getattr( )	locals( )	repr( )	zip( )
compile( )	globals( )	map( )	reversed( )	__import__( )
complex( )	hasattr( )	[✓ max( )]	round( )	

In [ ]:

```
width  = 5
height = 3.6
snps   = ['rs123', 'rs5487']
snp    = 'rs2546'
active = True
nums   = [2,4,6,8,4,5,2]

sum(nums)
```

More on operations

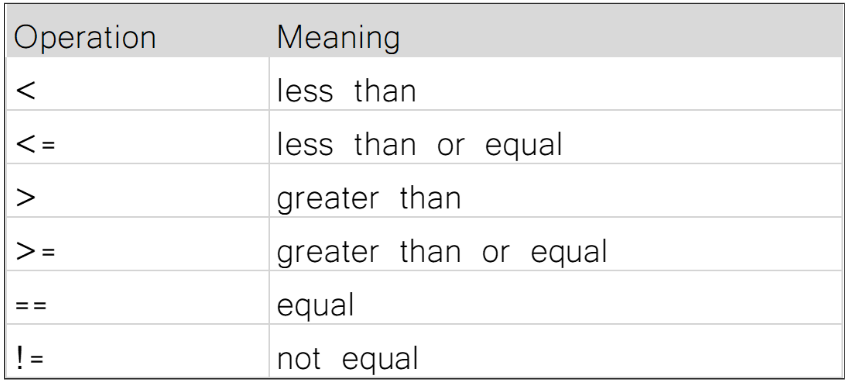


Operation	Result
x + y	sum of x and y
x − y	difference between x and y
x ∗∗ y	x to the power y
....	....
pow(x, y)	x to the power y
float(x)	x converted to float
int(x)	x converted to int!
len(z)	length of z if list
max(z)	maximum in list of z
min(z)	minimum in list of z

In [ ]:

```
x = 4
y = 3
z = [2, 3, 6, 3, 9, 23]

max(z)
```

Comparison operators



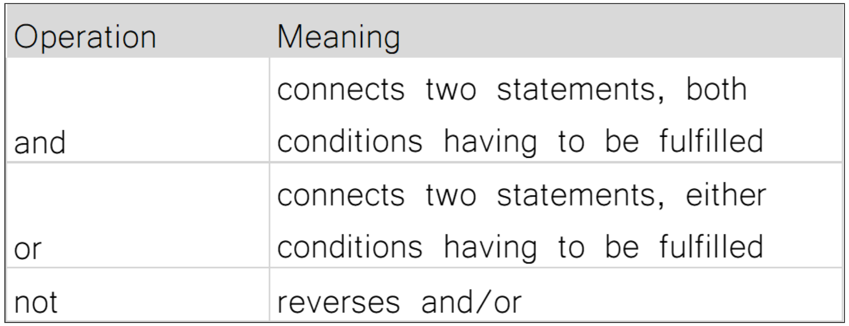
Operation	Meaning
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
!=	not equal

Can be used on int, float, str, and bool. Outputs a boolean.

In [ ]:

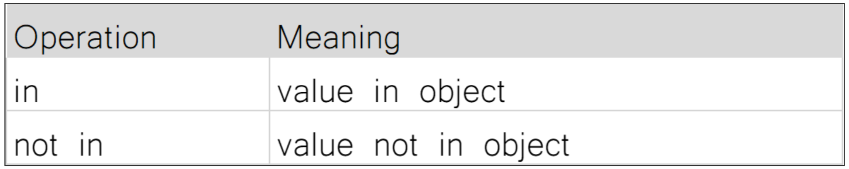
```
x = 5
y = 3

#x = 5.14
#y = 3.14

y + 2 == x
```

Logical operators



Operation	Meaning
and	connects two statements, both conditions having to be fulfilled
or	connects two statements, either conditions having to be fulfilled
not	reverses and/or

Membership operators

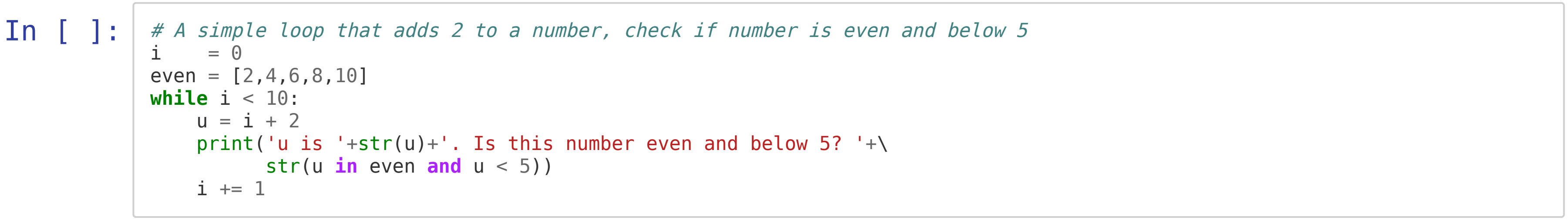


Operation	Meaning
in	value in object
not in	value not in object

In [ ]:

```
x = 2
y = 3

x == 2 and y == 5

#x = [2,4,7,3,5,9]
#y = ['a','b','c']

#23 in x
#4 in x and 'd' in y
```

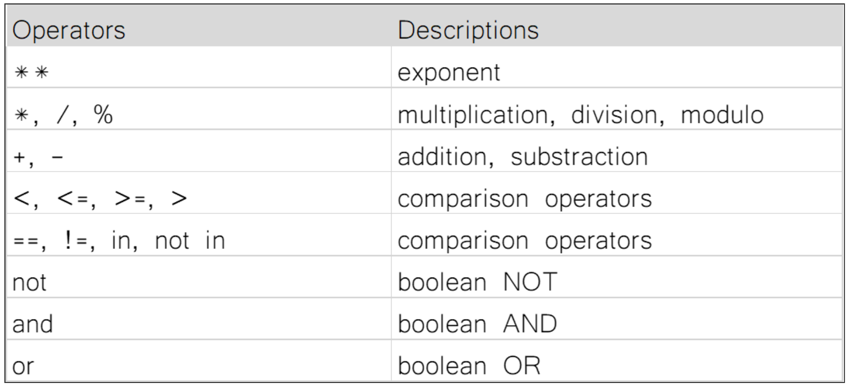
```
In [ ]: # A simple loop that adds 2 to a number and checks if the number is even
        i    = 0
        even = [2,4,6,8,10]
        while i < 10:
            u = i + 2
            print('u is '+str(u)+'. Is this number even? '+str(u in even))
            i += 1
```



```python
# A simple loop that adds 2 to a number, check if number is even and below 5
i    = 0
even = [2,4,6,8,10]
while i < 10:
    u = i + 2
    print('u is '+str(u)+'. Is this number even and below 5? '+\
          str(u in even and u < 5))
    i += 1
```

Order of precedence

There is an order of precedence for all operators:



Operators	Descriptions
**	exponent
*, /, %	multiplication, division, modulo
+, -	addition, substraction
<, <=, >=, >	comparison operators
==, !=, in, not in	comparison operators
not	boolean NOT
and	boolean AND
or	boolean OR

Word of caution when using operators

In [ ]:

```
x = 5
y = 7
z = 2
(x > 6 and y == 7) or z > 1

#x > 6 and (y == 7 or z > 1)
#(x > 6 and y == 7) or z > 1

#x > 4 or y == 6 and z > 3
#x > 4 or (y == 6 and z > 3)
#(x > 4 or y == 6) and z > 3
```

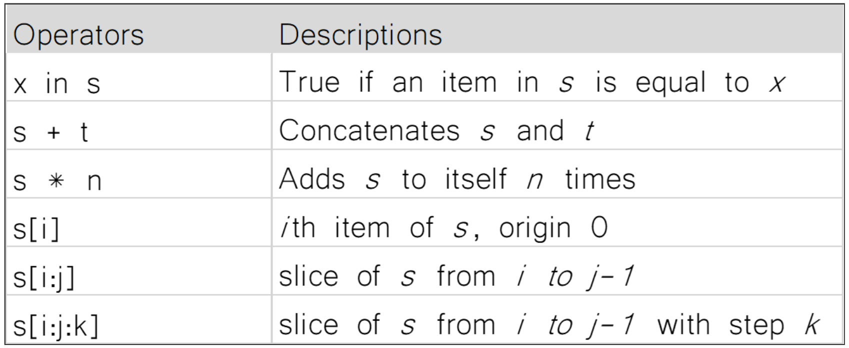
In [ ]:

```
# BEWARE!
x = 5
y = 8

x > 2 or xx == 6 and xxx == 6
x > 42 or (y < 8 and someRandomVariable > 1000)
```

Python does short-circuit evaluation of operators

More on sequences (For example strings and lists)

Lists (and strings) are an ORDERED collection of elements where every element can be accessed through an index.



Operators	Descriptions
x in s	True if an item in s is equal to x
s + t	Concatenates s and t
s ∗ n	Adds s to itself n times
s[i]	ith item of s, origin 0
s[i:j]	slice of s from i to $j-1$
s[i:j:k]	slice of s from i to $j-1$ with step k

In [ ]:

```
l = [2,3,4,5,3,7,5,9]
s = 'somelongrandomstring'

#s[0]
#s[0:4]
#s[0:4:2]
#s[0] = 'S'
```

Mutable vs Immutable objects

Mutable objects can be altered after creation, while immutable objects can't.

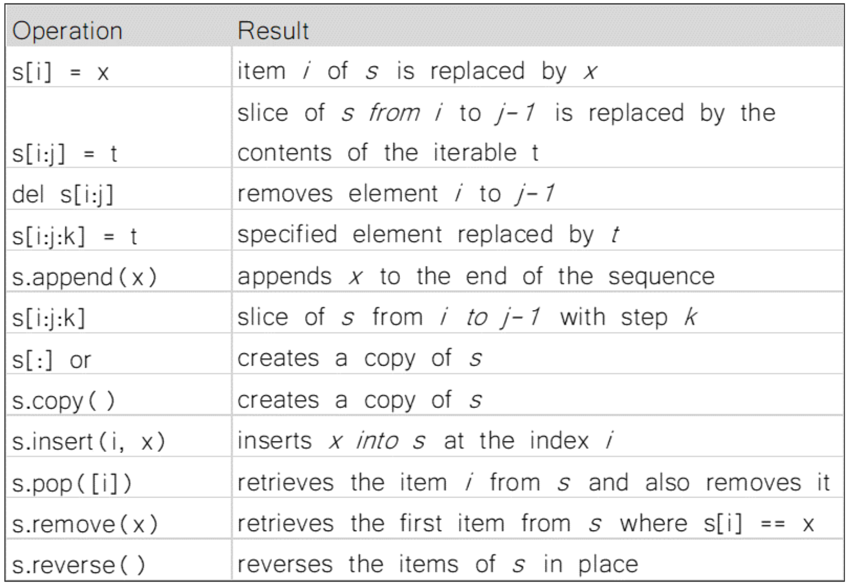
Immutable objects:

- `int`
- `float`
- `bool`
- `str`
- `tuple`

Mutable objects:

- `list`
- `set`
- `dict`

Operations on mutable sequences



Operation	Result
s[i] = x	item *i* of *s* is replaced by *x*
s[i:j] = t	slice of *s from i* to *j-1* is replaced by the contents of the iterable t
del s[i:j]	removes element *i* to *j-1*
s[i:j:k] = t	specified element replaced by *t*
s.append(x)	appends *x* to the end of the sequence
s[i:j:k]	slice of *s* from *i to j-1* with step *k*
s[:] or	creates a copy of *s*
s.copy()	creates a copy of *s*
s.insert(i, x)	inserts *x into s* at the index *i*
s.pop([i])	retrieves the item *i* from *s* and also removes it
s.remove(x)	retrieves the first item from *s* where s[i] == x
s.reverse()	reverses the items of *s* in place

In [ ]:

```
s = [0,1,2,3,4,5,6,7,8,9]
s.insert(5,10)

s
```

Summary

- The python standard library has many built-in functions regularly used
- Operators are used to carry out computations on different values
- Three types of operators; comparison, logical, and membership
- Order of precedence crucial!
- Mutable object can be changed after creation while immutable objects cannot be changed

→ **Notebook Day_1_Exercise_2 (~30 minutes)**

Loops in Python

In [ ]:

```
fruits = ['apple','pear','banana','orange']

print(fruits[0])
print(fruits[1])
print(fruits[2])
print(fruits[3])
```

In [ ]:

```
fruits = ['apple','pear','banana','orange']

for fruit in fruits:
    print(fruit)
```

Always remember to INDENT your loops!

Different types of loops

`For` loop

In [ ]:

```
fruits = ['apple','pear','banana','orange']

for fruit in fruits:
    print(fruit)
print('end')
```

`While` loop

In [ ]:

```
fruits = ['apple','pear','banana','orange']

i = 0
while i < len(fruits):
    print(fruits[i])
    i = i + 1
```

Different types of loops

`For` loop

Is a control flow statement that performs a fixed operation over a known amount of steps.

`While` loop

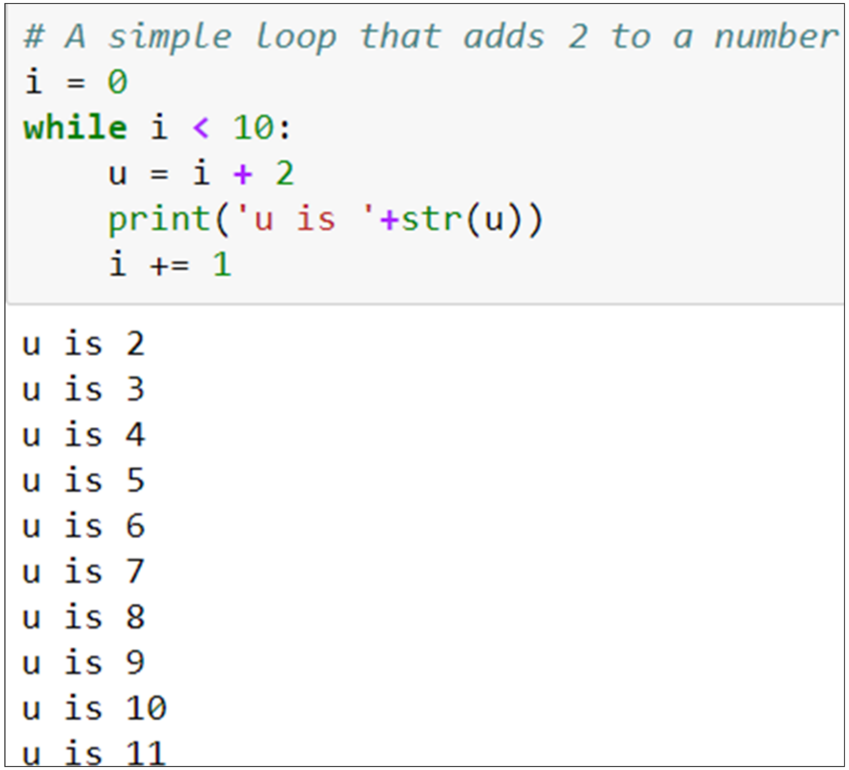
Is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition.

Which one to use?

`For` loops better for simple iterations over lists and other iterable objects

`While` loops are more flexible and can iterate an unspecified number of times

Example of a simple Python script

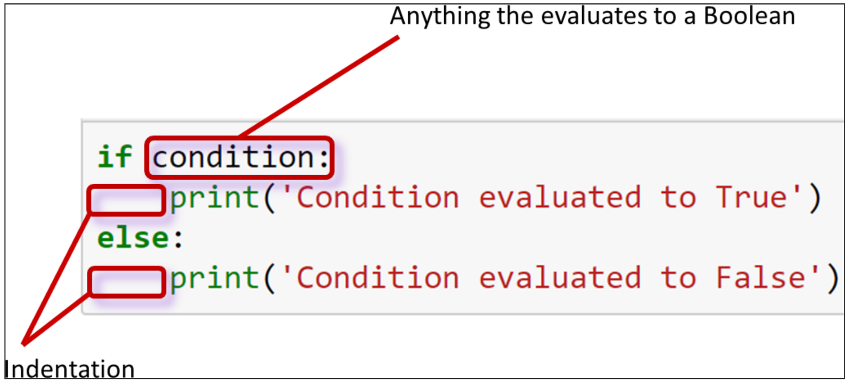


```python
# A simple loop that adds 2 to a number
i = 0
while i < 10:
    u = i + 2
    print('u is '+str(u))
    i += 1
```

```
u is 2
u is 3
u is 4
u is 5
u is 6
u is 7
u is 8
u is 9
u is 10
u is 11
```

→ **Notebook Day_1_Exercise_3 (~20 minutes)**

Conditional `if/else` statements



Anything the evaluates to a Boolean

```
if condition:
    print('Condition evaluated to True')
else:
    print('Condition evaluated to False')
```

Indentation

In [ ]:

```
shopping_list = ['bread', 'egg', 'butter', 'milk']

if len(shopping_list) > 2:
    print('Go shopping!')
else:
    print('Nah! I\'ll do it tomorrow!')
```

In [ ]:

```
shopping_list = ['bread', 'egg', 'butter', 'milk']
tired         = True

if len(shopping_list) > 2:
    if not tired:
        print('Go shopping!')
    else:
        print('Too tired, I\'ll do it later')
else:
    if not tired:
        print('Better get it over with today anyway')
    else:
        print('Nah! I\'ll do it tomorrow!')
```

This is an example of a nested conditional

Putting everything into a Python script

Any longer pieces of code that have been used and will be re-used SHOULD be saved

Two options:

- Save it as a text file and make it executable
- Save it as a notebook file

Examples

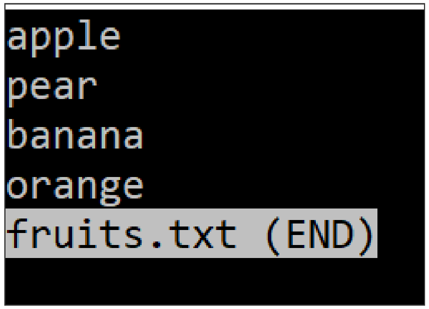
Things to remember when working with scripts

- Put *#!/usr/bin/env python3* in the beginning of the file
- Make the file executable to run with `./script.py`
- Otherwise run script with `python script.py`

Working on files

In [ ]:

```
fruits = ['apple','pear','banana','orange']

for fruit in fruits:
    print(fruit)
```



```
apple
pear
banana
orange
fruits.txt (END)
```

In [ ]:

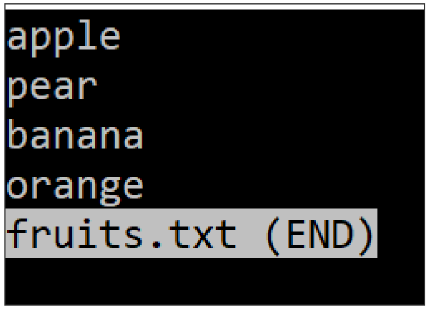
```
fh = open('../files/fruits.txt', 'r', encoding = 'utf-8')

for line in fh:
    print(line.strip())

fh.close()
```

Pause for additional useful methods:

`'string'.strip()`	Removes whitespace
`'string'.split()`	Splits on whitespace into list

In [ ]:

```
s  = 'an example string to split with whitespace in end   '
sw = s.strip()
sw
#l  = sw.split()
#l
#l  = s.strip().split()
#l
```

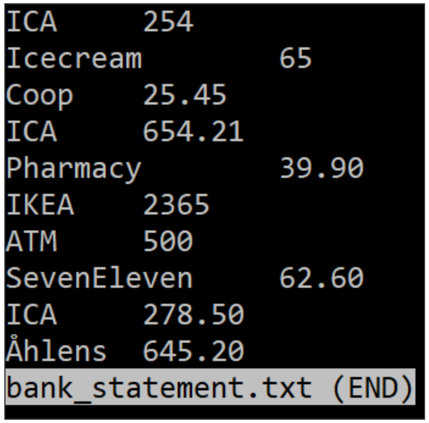


```
apple
pear
banana
orange
fruits.txt (END)
```

In [ ]:

```python
fh = open('../files/fruits.txt', 'r', encoding = 'utf-8')

for line in fh:
    print(line.strip())

fh.close()
```

Another example



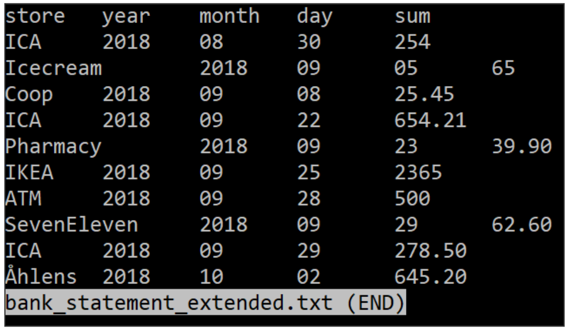
```
ICA	254
Icecream	65
Coop	25.45
ICA	654.21
Pharmacy	39.90
IKEA	2365
ATM	500
SevenEleven	62.60
ICA	278.50
Åhlens	645.20
bank_statement.txt (END)
```

How much money is spent on ICA?

In [ ]:

```
fh    = open("../files/bank_statement.txt", "r", encoding = "utf-8")

total = 0

for line in fh:
    expenses = line.strip().split()  # split line into list
    store    = expenses[0]           # save what store
    price    = float(expenses[1])    # save the price
    if store == 'ICA':               # only count the price if store is ICA
        total = total + price
fh.close()

print('Total amount spent on ICA is: '+str(total))
```

Slightly more complex...



```
store   year    month   day     sum
ICA     2018    08      30      254
Icecream        2018    09      05      65
Coop    2018    09      08      25.45
ICA     2018    09      22      654.21
Pharmacy        2018    09      23      39.90
IKEA    2018    09      25      2365
ATM     2018    09      28      500
SevenEleven     2018    09      29      62.60
ICA     2018    09      29      278.50
Åhlens  2018    10      02      645.20
bank_statement_extended.txt (END)
```

How much money is spent on ICA in September?

In [ ]:

```python
fh    = open("../files/bank_statement_extended.txt", "r", encoding = "utf-8")

total = 0

for line in fh:
    if not line.startswith('store'):
        expenses = line.strip().split()
        store    = expenses[0]
        year     = expenses[1]
        month    = expenses[2]
        day      = expenses[3]
        price    = float(expenses[4])
        if store == 'ICA' and month == '09':   # store has to be ICA and month september
            total = total + price
fh.close()

out = open("../files/bank_statement_result.txt", "w", encoding = "utf-8")   # open a file for writing the results to
out.write('Total amount spent on ICA in september is: '+str(total))
out.close()
```

Summary

- Python has two types of loops, `For` loops and `While` loops
- Loops can be used on any iterable types and objects
- `If/Else` statement are used when deciding actions depending on a condition that evaluates to a boolean
- Several `If/Else` statements can be nested
- Save code as notebook or text file to be run using python
- The function `open()` can be used to read in text files
- A text file is iterable, meaning it is possible to loop over the lines

→ **Notebook Day_1_Exercise_4**