Python Kick-off (Extended)

Dr Peet Morris
peet.morris@it.ox.ac.uk

iT Centre Learning

iT services

UNIVERSITY OF OXFORD

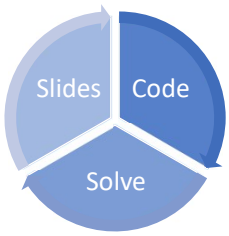
Feel free to contact me with questions about Python both during and after the course. I'm always happy to help.

What's Covered in this Kick-Off

- Main aim – to provide Python language basics, e.g…
  - *What is Python and how do I use it?*
  - *Options for writing and running Python programs (code editors and the like)*
  - *Python syntax and basic operations on objects*
    - *Iteration & Decision Making*
    - *Python's Fundamental Data Types*
      - *integers, floats, strings, lists, tuples, sets, dictionaries*
  - *The makeup of a Python program*
    - *Scripts, Modules, Packages*
    - *Python's Standard Library*
    - *Installing additional modules with Pip and PyPi*

This is an introduction to Python, not a course on how to use it once the basics have been learned.

How it Runs



- Repeated
  - *Slides – me talking, showing you stuff; you asking questions*
    - *Followed by*
  - *Together as a group we'll solve a problem by writing code*
    - *Followed by*
  - *Individual or group problem-solving (a sort of real-time homework)*

- At the end of the Kick-off
  - *You'll get a copy of the slides (inc notes)*
  - *Various post-course 'Challenge Problems' for you to work on to consolidate your knowledge (c/w code snippets and model answers)*

Python

- Created by Guido van Rossum (1991)
- Latest major version 3.10.5 (06/06/2022)
  - *What's New in 3.10*
- Interpreted & Compiled to Byte Code (CPython)
  - *Other special implementations available. For example, a version that runs on microcontrollers.*
- Dynamically typed

Python programs consist of a starting *script* (.py file), which may or may not **import** other scripts (.py files). Imported scripts are referred to not as scripts, but as *modules*.

A starting script is usually *interpreted*: in real-time, Python parses and acts upon the text (your *code*) in your script. However, imported modules are normally *compiled* (automatically) to a form of *compiled bytecode;* this is still however interpreted, but will run more efficiently.

When an imported module is compiled, a new file with a .pyc file is created to contain the bytecode. This normally happens just once (as module code rarely changes). However, if a change is detected, a module will be re-compiled down to bytecode.

.pyc files are stored in a subdirectory beneath the .py module file called __pycache__

Python is a *language specification*, and CPython is an implementation of that specification. It is the *gold standard* version, and, as it is created by the core developers (who are in charge of the specification), it will always be more up-to-date than any other versions.

Python – The Library, and other Packages

- Standard Modules/Packages (337)
  - *docs.python.org/3/py-modindex.html*
- Others
  - *pypi.python.org/pypi*
  - ≈270,000 packages/projects
    - *Graphical user interfaces*
    - *Web frameworks*
    - *Multimedia*
    - *Databases*
    - *Networking*
    - *Test frameworks*
    - *Automation*
    - *Web scraping*

- *System administration*
- *Scientific computing*
- *Text and NL processing*
- *Image processing*
- *Machine learning and AI*
- *Games development*
- *Home automation*
- *Financial*
- *...*

docs.python.org/3.10/library/index.html

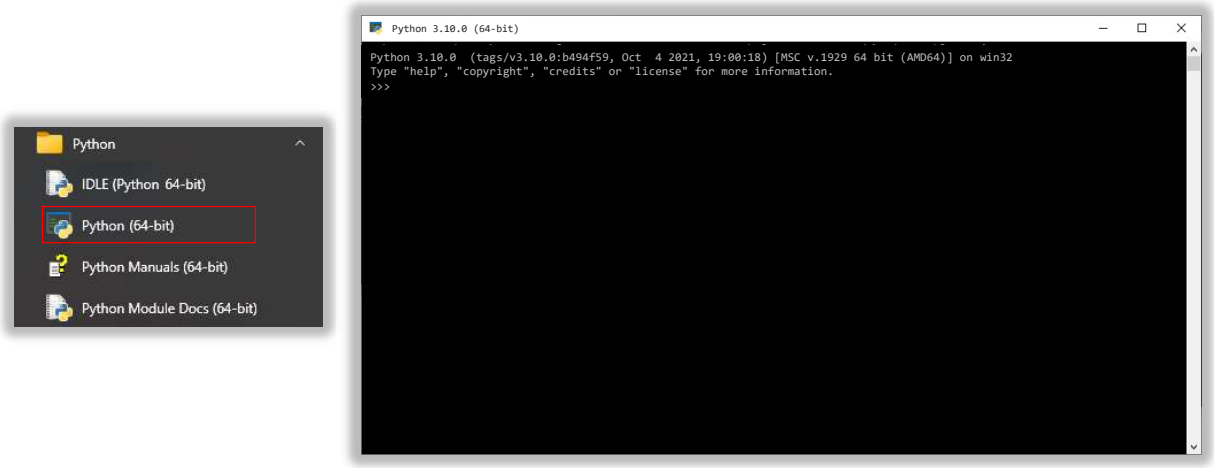
When you install Python (the interpreter) you will also install a library of extra modules referred to as the Python Standard Library. For example, the library contains modules to perform text processing, interacting with the host operating system, sending email, and very much more.

On top of this, https://pypi.python.org (PyPi) contains 1000s of extra modules and *packages* developed and shared by the Python community that can be searched, browsed and installed.

A *Package* is a group of modules that work together to perform a desired outcome.

https://packaging.python.org/en/latest/tutorials/installing-packages/

The Python REPL



The Python interpreter is variously called Python.exe (on Windows), or simply Python or a close variant on a Mac or Linux box where it is simply labelled as being executable (it doesn't need a '.exe' type of ending). On Windows you may not see the .exe ending as this is also normally hidden.

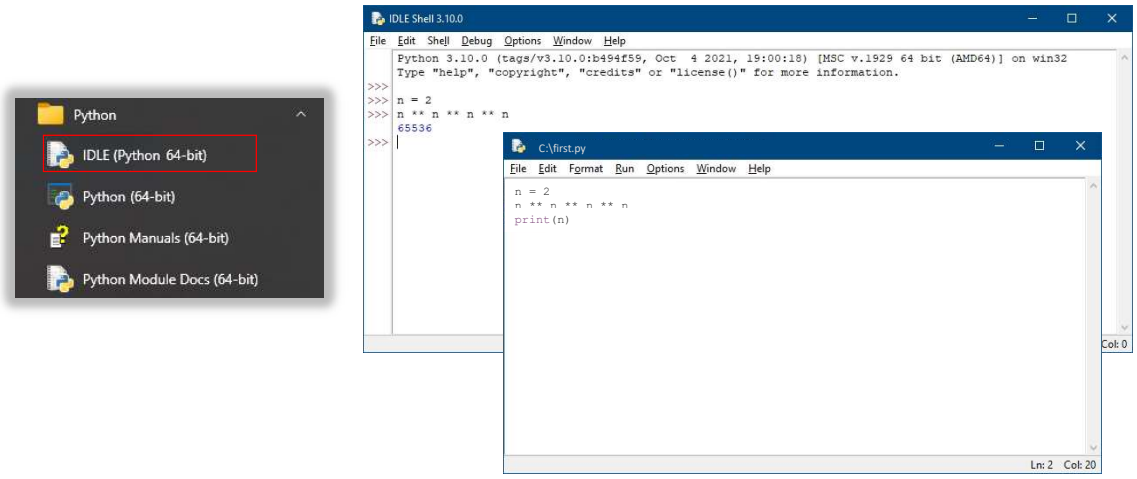
If you were to simply run Python, you would see what's called 'the repple' (REPL). REPL is an acronym which stands for Read; Evaluate; Print; Loop.

It is unusual to start Python like this unless you want to use it as a glorified calculator perhaps. However, you can write complete programs within the REPL, although you cannot save them away for later use.

Normally, the Python interpreter will be started automatically if you were to run a python script (by double-clicking it perhaps). In that case it would be equivalent to starting the interpreter and passing it a *command line argument* (the name of the script to run), e.g., **python.exe myscript.py** on Windows. If started in this way, the REPL environment isn't actually shown, just the output – if there is any – from your script/program.

(Eric) Idle

sourceforge.net/projects/idlex



Scripts etc are normally created using a smart editor – that understands the Python language syntax.

IDLE is such an editor/environment, and it comes with a CPython installation.

When developing programs this way it's normal to also see a REPL environment appear alongside the editor. This will show your program's output and will also allow you to inspect and debug your code.

Fully featured development environments are referred to as **IDE**s; *Integrated Development Environments*.

Variables have names, examples might be `x`, `y`, or `taxRate`. Variables always hold a value – in reality, they are a *pseudonym* for the value they hold. Values represented by - stored in - a variable may be changed; thus the name *variable*.

```
x = 10    # x is another name for the value 10.
x = x + 1 # x is another name for the value 11.
```

More on variables soon.

Installing a Python IDE

wiki.python.org/moin/IntegratedDevelopmentEnvironments

pyscripter

pycharm

code.visualstudio

Set the default configuration

anaconda.com

sublimetext.com

CodeSkulptor3

codeskulptor.org

repl.it

jupyter.org

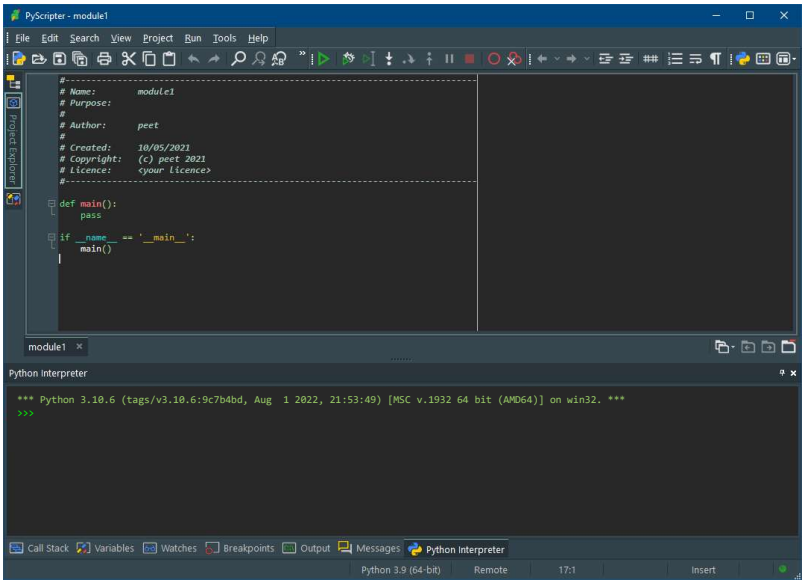
colab.research.google.com

Some examples of IDEs and Smart Editors. The Python.org website contains a complete list of each.

https://wiki.python.org/moin/IntegratedDevelopmentEnvironments

https://wiki.python.org/moin/PythonEditors

PyScripter (Windows)



PyScripter is my personal favourite IDE for Python work. However, it's probably not the best – but old dogs are reluctant to learn new tricks.

Using variables. $E = mc^2$

```
m = 1/1000                          # kilograms.

c = 299_792_458                     # metres per second.

e = m * (c ** 2)                    # joules.

eInTonOfTNT = 4.184 * 10 ** 9       # gigajoules.

print(round(e / eInTonOfTNT))       # energy in 1 gram of anything
                                    # equal to 21,481 tons of TNT.
21484
>>>
```

Variables, are not only useful if we want to change their value; we mainly use them to make our code clearer: by naming things, and splitting up complex statements, e.g., can you imagine making sense of the following without them?

```
print((1 / 1000 * 299_792_458 ** 2) / (4.184 * 10 ** 9))
```

FYI, a sugar cube weighs about 2.5 grams.

Built-in Data Types

Type	Mutable	Iterable
`int`	No	n/a
`bool`	No	n/a
`float`	No	n/a
`str`	No	n/a
`set`	Yes	Yes
`dict (immutable keys, values)`	*Yes*	Yes
`list (indexes numbers)`	Yes	Yes
`tuple`	No	Yes

See also: Collections – other high-performance container datatypes

As mentioned earlier; it's usual to use *variables* – usually with meaningful names, for example, we could have a variable called `basicTaxRate`. We can set an actual value into this using an *assignment operator*, followed by a *value*, e.g. `basicTaxRate = 20`. Now, basicTaxRate is really a pseudonym for the value set into it (which can change – thus the name *variable*)

You can imagine that when reading code it will probably be more obvious what the code is doing if we use sensible names:

```
netSalary = grossSalary ÷ 100 × basicTaxRate
```

Of course, `netSalary` and `grossSalary` will also have been created and set elsewhere. You can name your variables anyway you like really, the style used here is called *camelCasing*.

We use variables because if, say, the tax rate changed, we could reflect that throughout our code by simply changing a single line, e.g., **`basicTaxRate = 25`**.

The built-in and fundamental *types* of variables we can create are shown in the slide.

Operators

- Operators: `+ - / * // % **` *(% is modulus; // is integer division)*
- Bitwise: `& | ~ ^ << >>` *(treating numbers as binary bits)*
- Logical: `and or not` (*if this or that, and not the other*)
- Other: `in` (membership) `is` (identity)
- Comparison: `== != >= <= > <`

On the previous slide ÷ and × were used to show division and multiplication. Python has special symbols for these and other operations, e.g., `//` means integer division (the result will always be a whole number) and `/` is used for floating point or *real* division.

Integers can be treated as binary digits, e.g.,

```
n = 3      # 00000011 - 3 in binary.
n = n << 1 # 00000110 - bits shifted one to the left.  6 in binary.
print(n)   # Will output 6.
```

The logical operators allow us to build *Boolean expressions* that will evaluate to being either True or False, so that we can make decisions e.g.,

```
if isSunny and (isSummer or isSpring) and hour >= 17 then …
```

Decision Making

```
# value check.
#
# is x's value the same as n's value?

if x == n: # testing == results in either True or False

    do something

else: # x and n don't have the same value. How do they differ?

    do something else


                    if ____ and ____ and not ____ or ____:
```

Basic branching/decision-making is done using an `if ... else` construct.

```
if isSunny and (isSummer or isSpring) and hour >= 17:

            # Go for a walk?
else:

            # Have to work?
            # Perhaps we also need to check what day it is!
```

Note the layout and the use of colons.

You might wonder about the order in which operators are applied, this is called Operator Precedence:

https://docs.python.org/3/reference/expressions.html#operator-precedence

Decision Making

```
if httpStatus == 400:
    print("Bad request")

elif httpStatus == 401 or httpStatus == 403: # elif means 'else if'
    print("Forbidden")

elif httpStatus == 404:
    print("Not found")

elif httpStatus == 418:
    print("I'm a teapot!")

else: # Nothing else matched httpStatus' value, so do this.
    print("Something went wrong, but who knows what!")
```

Maybe there are possibly more subtle states that we have to check for? In that case we have `elif` – meaning *else if*.

Note that, say, `httpStatus` had the value 401, that as soon as we output '`Forbidden`' we don't check `httpStatus` against any other possible values: as soon as we find a match, we're done checking.

Decision Making

```
if httpStatus == 400:
    print("Bad request")

elif httpStatus in (401, 403): # (401, 403) a 'Tuple' containing two values.
    print("Forbidden")

elif httpStatus == 404:
    print("Not found")

elif httpStatus == 418:
    print("I'm a teapot!")

else:
    print("Something went wrong, but who knows what!")
```

A more *Pythonic* way to check for either 401 or 403, is to see whether `httpStatus`' *value* is to also be found in the *tuple* containing the values 401 and 403.

More on Tuples later.

Decision Making – `match`, new in 3.10.0

```
match httpStatus:

    case 400:
        print("Bad request")

    case 401 | 403:
        print("Forbidden")

    case 404:
        print("Not found")

    case 418:
        print("I'm a teapot!")

    case _:
        print("Something went wrong, but who knows what!")
```

Much more powerful than this.

Introduced in Python 3.10, the `match` and `case` keywords may be used to check `httpStatus` like this.

Note the use of `|` (looks like the bitwise OR operator) is used to test for 401 *or* 403.

Iteration/Repetition

```
n = 5
i = 1

# Or
# n = 5; i = 1
# n, i = 5, 1

while i <= n:  # while True
               #
    print(i)   #     do this
               #
    i = i + 1  #     and this

print('Done')
```

```
for i in [1, 2, 3, 4, 5]: # a 'List'.

    print(i)

print('Done')


n = 5

for i in range(1, n + 1):

    print(i)

print('Done')
```

The only ways to iterate in Python are using `while` or `for`.

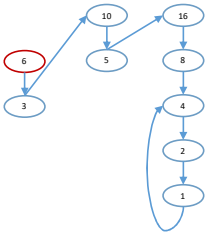
The *body* of the `while` loop here will run whenever it is True that `i`'s value is less than or equal to `n`'s value. Because `i` is being altered within the body of the loop, there will come a time when `i`'s value will be the same as `n`'s. At that point, the loop exits and the next line of code will be interpreted and the word 'Done' will be output.

The `range()` *function* creates a *range object*, which is a sort of *generator*. Here, `range()` will, when asked, *generate* the values `1` through to `5`, one at a time. The `for i in` will cause the sort-of-generator to run, setting each value generated into `i`. On each new value set into `i`, the *body* (indented part) of the *for loop* is entered and code found there is run. Once the sort-of-generator `range` is exhausted, the loop exits.

A generator only actually creates values as and when they are needed. The *list* equivalent code on the slide creates all of the values ahead of time. Not a problem here, but imagine say ranging over a billion values.

Problem Work-Through

- Write a program that continually re-calculates a value in a *loop*; stopping only if the calculated value is ever equal to 1:
  - Rules:
    - 1. Prompt for an arbitrary positive whole number.
    - 2. If the number's value is 1, the program terminates
    - 3. Otherwise, the program outputs the current value, and then checks whether the value is odd or even
      - If it is even, the program sets the current value to a new one by dividing it by 2, otherwise the program updates the current value by multiplying it by 3 and adding 1
    - Return to step 2.




```
# input() returns characters, int() converts suitable characters into
# a value, abs() makes sure that it's positive.
#
# We can nest functions – the output of one becomes the input of
# another.
#
n = abs(int(input('Enter a positive whole number: ')))

while n != 1:

    print(n)

    # n % 2 will either be 0 or 1. If 1, we know n was odd.
    #
    if n % 2:
        n = n * 3 + 1
    else:
        n = n // 2
```

Individual or Group Problem Solving

Real-time Problem (choose 1)

- Write a program that loops through *n* values, 1 – 100:
  - *if n is exactly divisible by 3, output 'fizz'*
  - *if n is exactly divisible by 5, output 'buzz'*
  - *if n is exactly divisible by both 5 and 3, output 'fizzbuzz'*
  - *if none of the above, just output n's value*
- Modify the 3*n* + 1 code:
  - *Modify the code so as to count the number of loops it makes before stopping, e.g., an input value of 100 decays to 1 in 26 loop-cycles*
  - *Try different starting values, what is your personal record for the number of cycles taken for a particular input?*
  - *Can you find a starting value so that the code never completes (n never goes to 1)?*

Fizzbuzz – a solution

```
for n in range(1, 100 + 1):

    if n % 3 == 0 and n % 5 == 0: print('fizzbuzz')

    elif n % 3 == 0: print('fizz')

    elif n % 5 == 0: print('buzz')

    else: print(n)
```

The initial `if` could have used parentheses to better show how the expressions will be evaluated: `if (n % 3 == 0) and (n % 5 == 0)`.

Any non-zero remainder will be consider `True`, so we could have written the same thing like this:

`if not n % 3 and not n % 5: print('fizzbuzz')`, again, we could also use parentheses to help readers of our code.

```
if (not (n % 3)) and (not (n % 5)): print('fizzbuzz')
```

Of course, the code here is dependent on order, e.g., we have to test for the case where `n` is exactly divisible by both `5` and `3` first.

Fizzbuzz – a solution

```
for n in range(1, 100 + 1):

    if n % 15 == 0: print('fizzbuzz')

    elif n % 3 == 0: print('fizz')

    elif n % 5 == 0: print('buzz')

    else: print(n)
```

If n is exactly divisible by both 5 and 3, we can combine the two tests into one and just test if n is exactly divisible by 15.

Fizzbuzz – a solution

```
for n in range(1, 100 + 1):

    s = '' # empty string, length is zero.

    if n % 3 == 0: s += 'fizz' # same as s = s + 'fizz'.

    if n % 5 == 0: s += 'buzz'

    if len(s) == 0: s = str(n) # s is still empty? Set it to string version of n.

    print(s)
```

This puzzle can be solved in so many different ways (as can more-or-less anything that can be solved via a bit of programming).

For example, here are a couple of slightly off the wall solutions to 'fizzbuzz' using some other Python features and syntax.

```
print('\n'.join(["fizzBuzz" if not x % 3 and not x % 5 else "fizz" if not x % 3 else "fuzz" if not x % 5 else str(x) for x in range(1, 101)]))
```

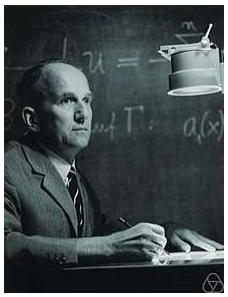
```
print("\n".join(["fizz" * (i % 3 == 0) + "buzz" * (i % 5 == 0) or str(i) for i in range(1, 101)]))
```

The skill is in choosing the right solution to a problem of course; and that may depend on many factors. E.g., does 'right' mean 'optimal' (speed), or simply 'clear' perhaps? I think we'd argue that the two solutions just above are less clear than the one on the slide?

By the way, the * in the second example is used to produce any number of copies of a string, e.g., "fizz". So `"fizz" * 2` results in `"fizzfizz"` and `"fizz" * 0` results in `""`.

$3n + 1$ Problem (the Collatz Conjecture)

```
n = abs(int(input('Enter a whole positive integer value')))

while True:

    print(n)

    if n == 1:
        break # we're done, breaks out of the loop.

    if n & 1: # n is Odd. Could also use if n % 2 == 1, or simply, if n % 2:

        n *= 3 # same as n = n * 3.
        n += 1 # same as n = n + 1.

    else: # n is Even.

        n //= 2 # same as n = n // 2.
```



Lothar Collatz

Paul Erdős said, "Mathematics may not be ready for such problems".

Further reading

```
for n in range(11):

    # using an 'f string'; print n, left-padded with 1 space if necessary.
    #
    # using bin(), convert n to a binary string; strip off leading '0b'
    # using [2:], then, right-pad the result to 8 characters in width
    # padded with zeros if necessary.
    #
    print(f'{str(n).ljust(2, " ")} -> {bin(n)[2:].rjust(8, "0")}')
```

```
0  -> 00000000 # Note that for every odd value of n, its right-most,
1  -> 00000001 # least-significant bit is '1'.
2  -> 00000010
3  -> 00000011
4  -> 00000100
5  -> 00000101
6  -> 00000110
7  -> 00000111
8  -> 00001000
9  -> 00001001
10 -> 00001010
```

`n & 1` takes the values of `n` and `1` and tests to see if their *least significant bits* and the same, consider an 8 bit representation and testing where `n` is `3`.
`n = 00000011, 1 = 00000001. n & 1` results in `1` as both `1` and `3` have their lower bit *set*. They are each made up of a single `1`.

`1` is considered `True`, as is **any value** other than zero. Thus, `if n & 1` is all that's needed, rather than `if n & 1 == 1`. The latter may be clearer for some readers of course.

Outputting values

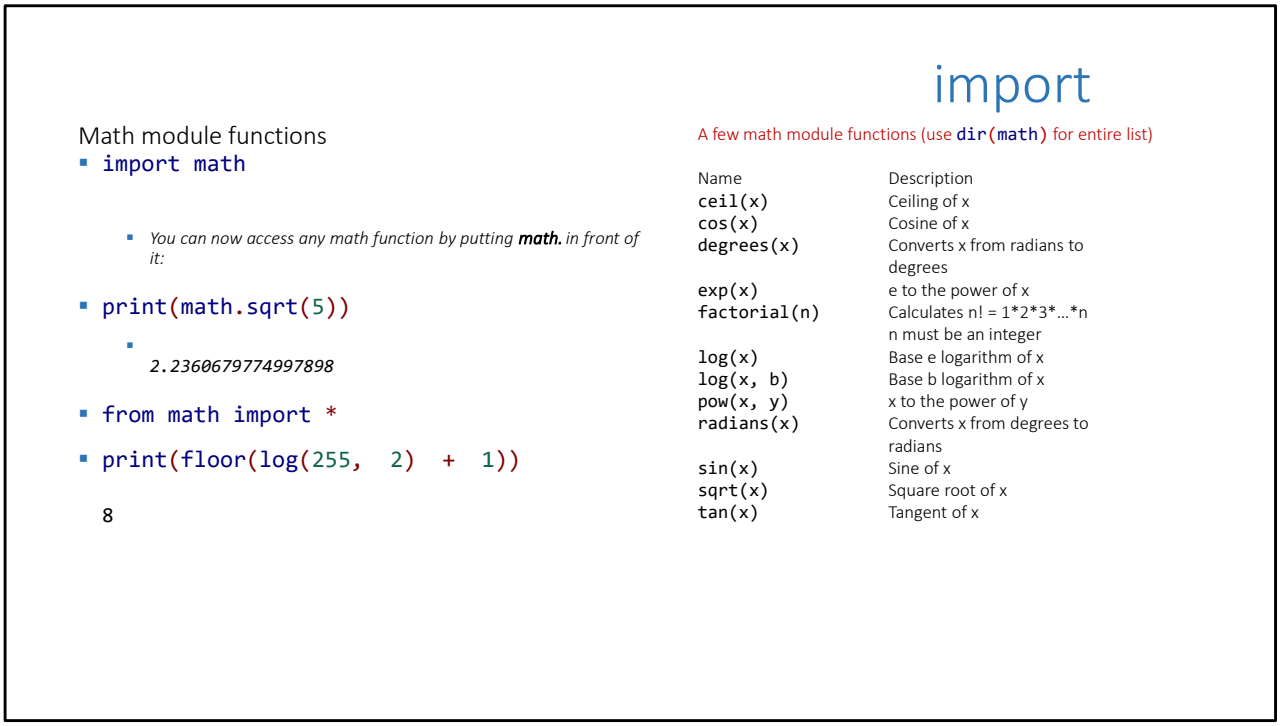
```
print(x, y)

print("x is ", end = ''); print(x, end = ''); print(", y is ", end = ''); print(y)

print("x is ", x, ", y is ", y)

print("x is " + str(x) + ", y is " + str(y))

print("x is %d, y is %.1f" % (x, y)) # printf style, from C/C++

print("x is {}, y is {}".format(x, y)) # format >= Python 2.6

print("x is {0}, x + y is {0} + {1} = {2}".format(x, y, x + y))

print(f"x is {x}, x + y is {x} + {y} = {x + y}") # f-string >= Python 3.6
```

old ways to format output

new way to format output

The *printf style* (a C language construction) is original Python, and was superseded by the easier, and in all ways superior, *format style* back in 2008 (*format* was retro-fitted into Python 2.6 from Python 3.0). In the same manner, the format style introduced in 3.0 is now superseded in 3.6 and later versions by *f-strings*.

If you are using Python 3.6 or later, use f-strings.



If you want to know where a module *lives*, import it, and then use `help()` like this:

```
import random
help(random)
```

At the bottom of the output you'll see something like:

```
FILE
       c:\program files\python310\lib\random.py
```

However, some modules are built in to the interpreter (over 60 currently). The `math` module is like this. If you try the trick above on that you'll see:

```
FILE
    (built-in)
```

If you want a full list of the available modules you have installed, use `help('modules').` If you want to know which are 'built in', `import sys`, then use `print(sys.builtin_module_names)`. Lastly, if you want to know where Python looks for non built-in modules, use `import sys`, then `print(sys.path)`

Built-in Data Types

Type	Mutable	Iterable	Subscriptable
int	No	n/a	n/a
bool	No	n/a	n/a
float	No	n/a	n/a
str	No	Yes	Yes
list	Yes	Yes	Yes
dict	Yes *(values)*	Yes	Yes *(keys)*
tuple	No *(ish)*	Yes	Yes
set	Yes	Yes	No

See also: Collections - High-performance container datatypes

Tuples are *immutable*, but things they contain may be *mutable*.

```
n = 10

t = tuple([n, []])

print(t) # outputs (10, [])

# t[0] = t[0] + 1 # error, cannot change n.

# t[1] = {} # error, second item can't be changed.

t[1].append(42) # all ok, adding an item to the empty list.

print(t) # outputs (10, [42])
```

List – ordered, indexed

- Created using `[ ]` or `list()`, e.g.,
  - `l = [0, '1', 2]` `print(l)` → `[0, '1', 2]`
  - `l = list('012')` `print(l)` → `['0', '1', '2']`
  - `l = [int(n) for n in '012'] # a list comprehension, [0, 1, 2].`
- Iterable
  - `for n in l: print(n)`
- Mutable / sliceable
  - `l.append(4)`
  - `l = l[1:] # Slicing.`
- Indexable
  - `print(l[1])`

Yesterday we saw `range()` compared to a list of values when looking at `for` loops:

```
for i in [1, 2, 3, 4, 5]: # a 'List'.
    print(i)
```

Using a *comprehension*, we could build the list automatically:

```
for i in [n for n in range(1, 6)]: # a 'List'.
    print(i)
```

You wouldn't ever do this in real life, but I hope it helps in your understanding.

Dictionaries – ordered (3.6), indexed

- Created using `{}` or `dict()`, e.g.,
  - `d = {0:'1', 1:'2', 2:'3'}` `print(d)` → `{0:'1', 1:'2', 2:'3'}`
    `print(list(d))` → `[0, 1, 2] # keys.`
  - `d = {a:str(b) for a, b in enumerate(range(1, 4))} # dictionary comprehension.`
- Iterable
  - `for n in d: print(n) # prints keys. print(n, d[n]) prints keys and values.`
- Mutable
  - `d[len(d)] = '4' # changes the 2 key's value to '4'.`
- Indexable
  - `print(d[1]) # prints the value associated with the key of 1, which is '2'.`

```
for i, j in enumerate(range(100, 106)):

    print(i, j)
```

`i` tells you which loop context you're operating in (0 – 5), `j` tells you the value taken from the `range` (in this case)

```
0 100
1 101
2 102
...
```

```
for i, j in enumerate('fizzbuzz'):

    print(i, j)
```

```
0 f
1 i
2 z
...
```

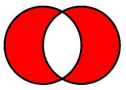
tuple – ordered, indexed

- Created using `( )` or `tuple()`, or `,` e.g.,
  - `t = (0, '1', 2)` `print(t)` → `(0, '1', 2)`
  - `t = 0, '1', 2` `print(t)` → `(0, '1', 2)`
  - `t = tuple('012')` `print(t)` → `('0', '1', '2')`
  - `t = (int(n) for n in '012') # a generator.`
- Iterable
  - `for n in t: print(n)`
- Immutable / sliceable
  - `t = t[1:] # Slicing.`
- Indexable
  - `print(t[1])`

What's the difference between `range()` and a *tuple generator*?

https://treyhunner.com/2018/02/python-range-is-not-an-iterator/

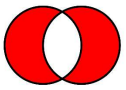
Set – unordered, unindexed



- Created using `{?}` or `set()`, e.g.,
  - `s = {0, '1', 2}` `s.add(2)` `print(s)` → `{0, '1', 2}`
  - `s = set('012')` `print(s)` → `{'0', '1', '2'}`
  - `s = {int(n) for n in '012'}` `print(3 not in s) # same for any iterable.`
- Iterable
  - `for n in s: print(n)`
- Mutable
  - `s.add(4); s.remove(3)`
- Set operations
  - `s.difference(k);` `s - k`
  - `s.intersection(k);` `s & k`
  - `s.union(k);` `s | k`
  - `s.symmetric_difference(k);` `s ^ k`
  - `s.subset(k)`
  - `s.superset(k)`

Sets are unordered – although it may appear that they are if you output them. So, it doesn't make any sense to index into them using square brackets.

Set



- `s.difference(k);` `s - k`
- `s.intersection(k);` `s & k`
- `s.union(k);` `s | k`
- `s.symmetric_difference(k);` `s ^ k`

```
s = set([1, 2, 3]); k = set([3, 4, 5])
print(s)          # {1, 2, 3}
print(k)          # {3, 4, 5}
print(s - k)      # {1, 2}
print(s & k)      # {3}
print(s | k)      # {1, 2, 3, 4, 5}
print(s ^ k)      # {1, 2, 4, 5} … {(s - k) U (k - s)}
```

```
s = set([1, 2, 3]); k = set([3, 4, 5])

print(s, end = ' ')                 # {1, 2, 3} {3, 4, 5}
print(k)

print(s - k, end = ' ')             # {1, 2} {1, 2}
print(s.difference(k))

print(s & k, end = ' ')             # {3} {3}
print(s.intersection(k))

print(s | k, end = ' ')             # {1, 2, 3, 4, 5, 6} {1, 2, 3, 4, 5, 6}
print(s.union(k))

print(s ^ k, end = ' ')             # {1, 2, 4, 5} {1, 2, 4, 5}
print(s.symmetric_difference(k))
```

`print()` normally ends its output by moving the cursor to the beginning of a new line. Called inserting a carriage-return and new-line (if you know what one is; think typewriters – also, if you know what one is!)

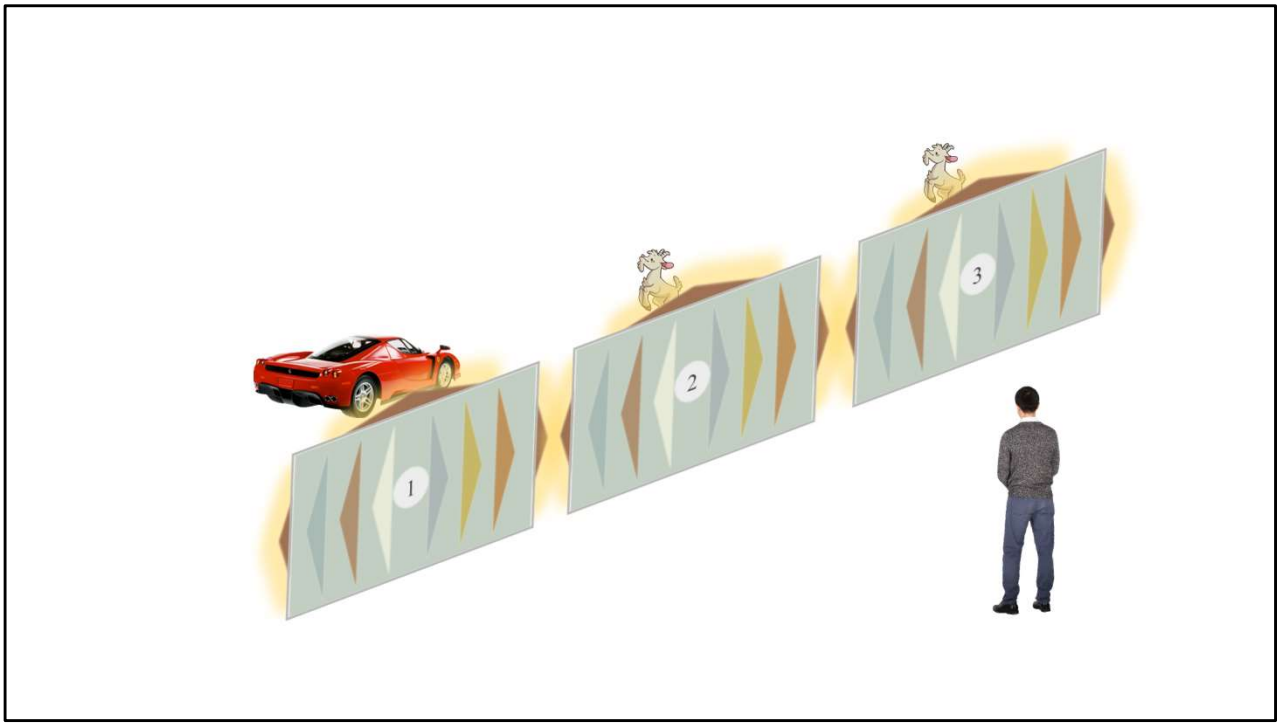
We can prevent that by adding a special trailing specification. Here that's just to insert a space; the next `print()`'s output will appear after that space.

Another way of saying that is, if excluded, the trailing expression defaults to be `'\n'` – which means output a carriage-return and new-line.

Problem Walk-Through

The Monty Hall Problem

Please see: http://cslab.com/monty/



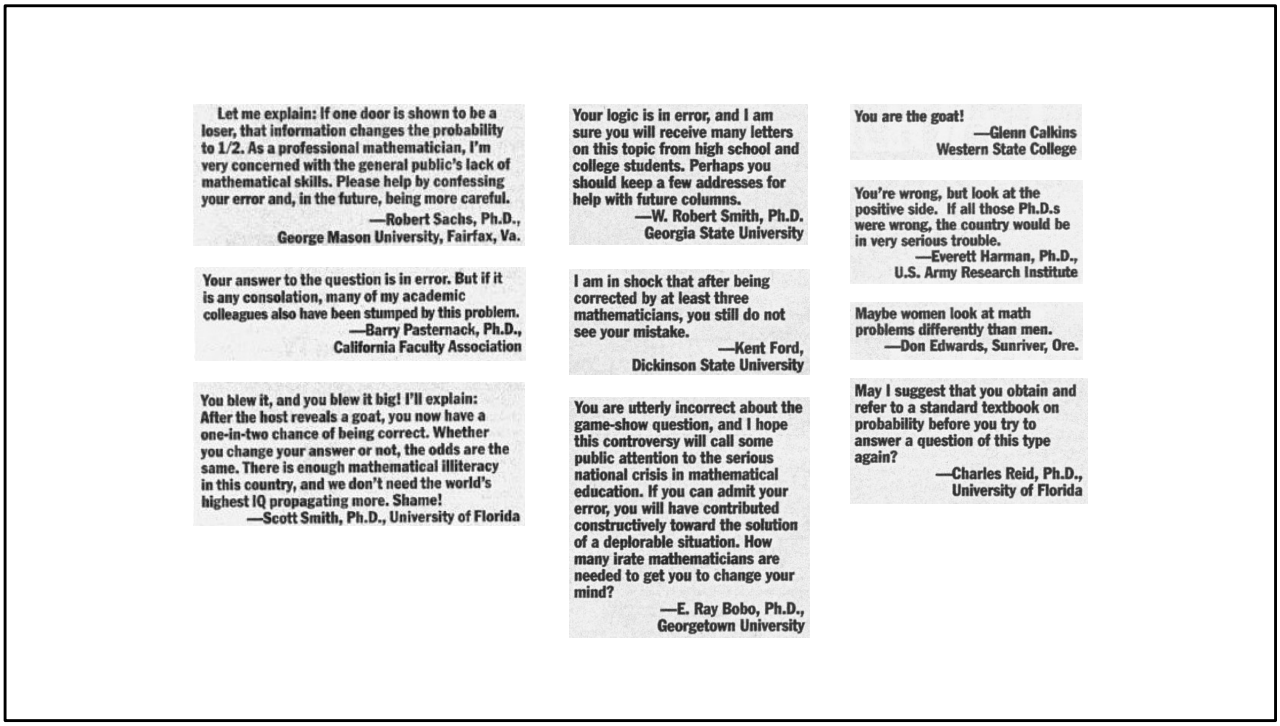
The host knows how the car and the goats are distributed.

Ask Marilyn™

BY MARILYN VOS SAVANT

Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?
—Craig F. Whitaker, Columbia, Md.

Yes; you should switch. The first door has a one-third chance of winning, but the second door has a two-thirds chance. Here's a good way to visualize what happened. Suppose there are a *million* doors, and you pick door No. 1. Then the host, who knows what's behind the doors and will always avoid the one with the prize, opens them all except door #777,777. You'd switch to that door pretty fast, wouldn't you?



Let me explain: If one door is shown to be a loser, that information changes the probability to 1/2. As a professional mathematician, I'm very concerned with the general public's lack of mathematical skills. Please help by confessing your error and, in the future, being more careful.
—Robert Sachs, Ph.D., George Mason University, Fairfax, Va.

Your answer to the question is in error. But if it is any consolation, many of my academic colleagues also have been stumped by this problem.
—Barry Pasternack, Ph.D., California Faculty Association

You blew it, and you blew it big! I'll explain: After the host reveals a goat, you now have a one-in-two chance of being correct. Whether you change your answer or not, the odds are the same. There is enough mathematical illiteracy in this country, and we don't need the world's highest IQ propagating more. Shame!
—Scott Smith, Ph.D., University of Florida

Your logic is in error, and I am sure you will receive many letters on this topic from high school and college students. Perhaps you should keep a few addresses for help with future columns.
—W. Robert Smith, Ph.D. Georgia State University

I am in shock that after being corrected by at least three mathematicians, you still do not see your mistake.
—Kent Ford, Dickinson State University

You are utterly incorrect about the game-show question, and I hope this controversy will call some public attention to the serious national crisis in mathematical education. If you can admit your error, you will have contributed constructively toward the solution of a deplorable situation. How many irate mathematicians are needed to get you to change your mind?
—E. Ray Bobo, Ph.D., Georgetown University

You are the goat!
—Glenn Calkins Western State College

You're wrong, but look at the positive side. If all those Ph.D.s were wrong, the country would be in very serious trouble.
—Everett Harman, Ph.D., U.S. Army Research Institute

Maybe women look at math problems differently than men.
—Don Edwards, Sunriver, Ore.

May I suggest that you obtain and refer to a standard textbook on probability before you try to answer a question of this type again?
—Charles Reid, Ph.D., University of Florida

92% of Americans thought Marilyn wrong.

Problem Work-Through

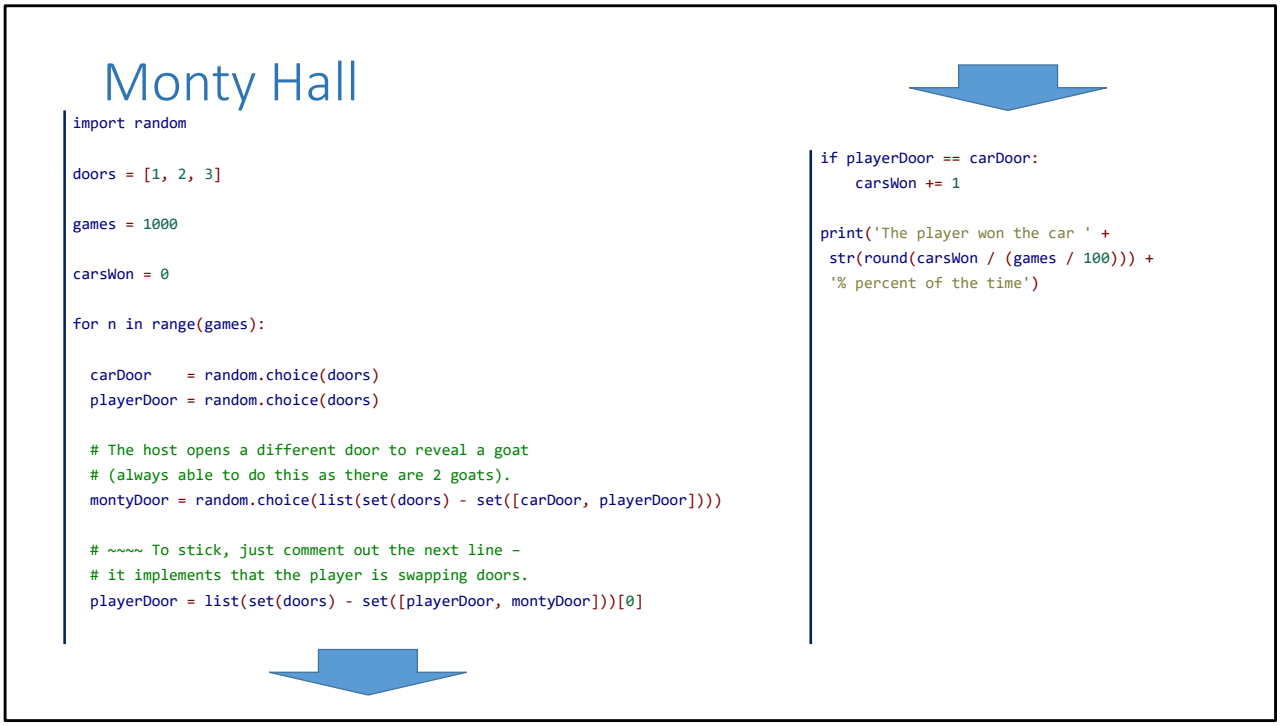
- To test Marilyn's assertion, let's write a simulation of 'The Monty Hall Problem'.
  - *Monte Carlo Simulation (inferential statistics)*

Even when given explanations, simulations, and formal mathematical proofs, many people still did not accept that switching is the best strategy. Indeed, Paul Erdős, one of the most prolific mathematicians in history, remained unconvinced until he was shown a computer simulation demonstrating vos Savant's predicted result.

https://web.archive.org/web/20140413131827/http://www.decisionsciences.org/DecisionLine/Vol30/30_1/vazs30_1.pdf

Monte Carlo methods, experiments or simulations, are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results.

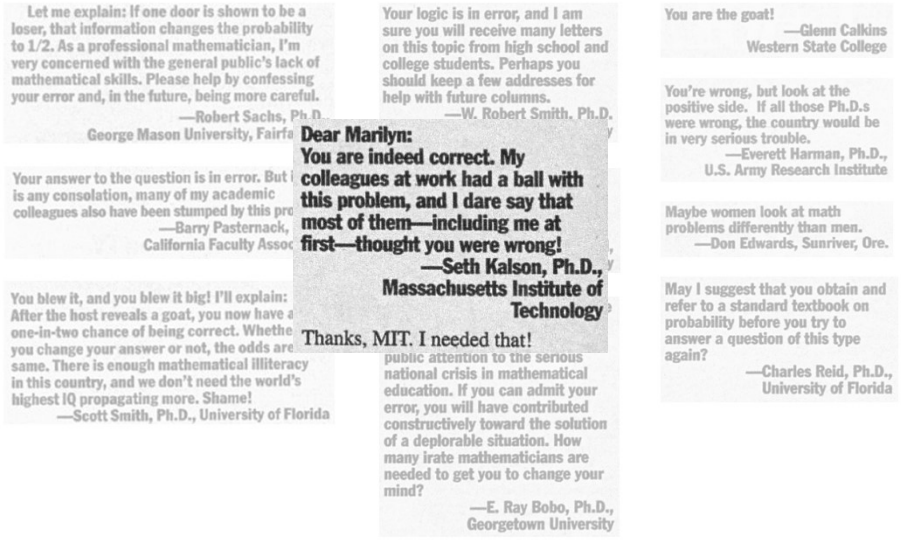
The underlying concept is to use randomness to solve problems that might be deterministic in principle but that have a number of variables and/or states that make that determinism more difficult to calculate and keep abreast of. They are often applied to probability problems: in which the human mind often comes to incorrect conclusions from often simple premises and seemingly simply and obvious questions.



Some attendees will be familiar with R (https://www.r-project.org).

An example of the equivalent R code to solve the problem, complete with an explanation, may be found here:

https://bookdown.org/danbarch/psy_207_advanced_stats_I/markov-chain-monte-carlo-methods.html#lets-make-a-deal



See cslab.com/monty for more.

92% of Americans weren't as smart as Marilyn.

Individual or Group Problem Solving

Real-time Problem

- Modify the 3n + 1 code:
  - *If we didn't break out of the loop when we get to 1, we would loop endlessly over 4, 2, 1.*

    Alter your code to use a Python `set` *to detect that we've previously seen an output, and break out of your loop when that's seen.*
  - *Can you find other (hailstone) sequences by altering the algorithm slightly, i.e., those not terminating in 4, 2, 1? Hint, you'll need to start with negative starting values and look no further than, say, -20.*

    Can you do this programmatically (test a range of inputs consecutively in an outer loop)?

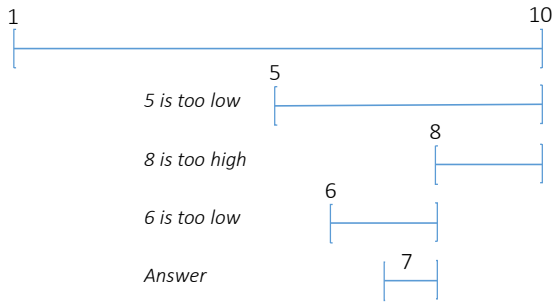
- Use a *list comprehension* to create a `list` of the squares of the integers 1 – 10 inclusive.
  - *Then; a bit more complex – use* `enumerate()` *to build a* `list` *of two-element* `lists` *– the thing being squared, and the squared result, e.g.,*

```
[[1, 1],      # First enumeration, 1² is 1
 [2, 4],      # Second enumeration 2² is 4.
 [3, 9],      # ...
 [4, 16],
 [5, 25],
 [6, 36],
 [7, 49],
 [8, 64],
 [9, 81],
 [10, 100]]
```

```
n = abs(int(input('Enter a positive whole number: ')))

loops = 0

s = set() # s is an empty set.

while True: # infinite loop.

    print(n)

    # is n found in set s? If yes, we're done. Break out of loop.
    if n in s:
        break

    s.add(n) # Not seen n previously, so add n as a member of s.

    # if n is odd, its least significant bit is set, n & 1 will be 1.
    #
    if n & 1:
        n = n * 3 + 1
    else:
        n = n // 2

print(f'Loops: {loops}') # An 'f-string'.
```

Real-time Problem

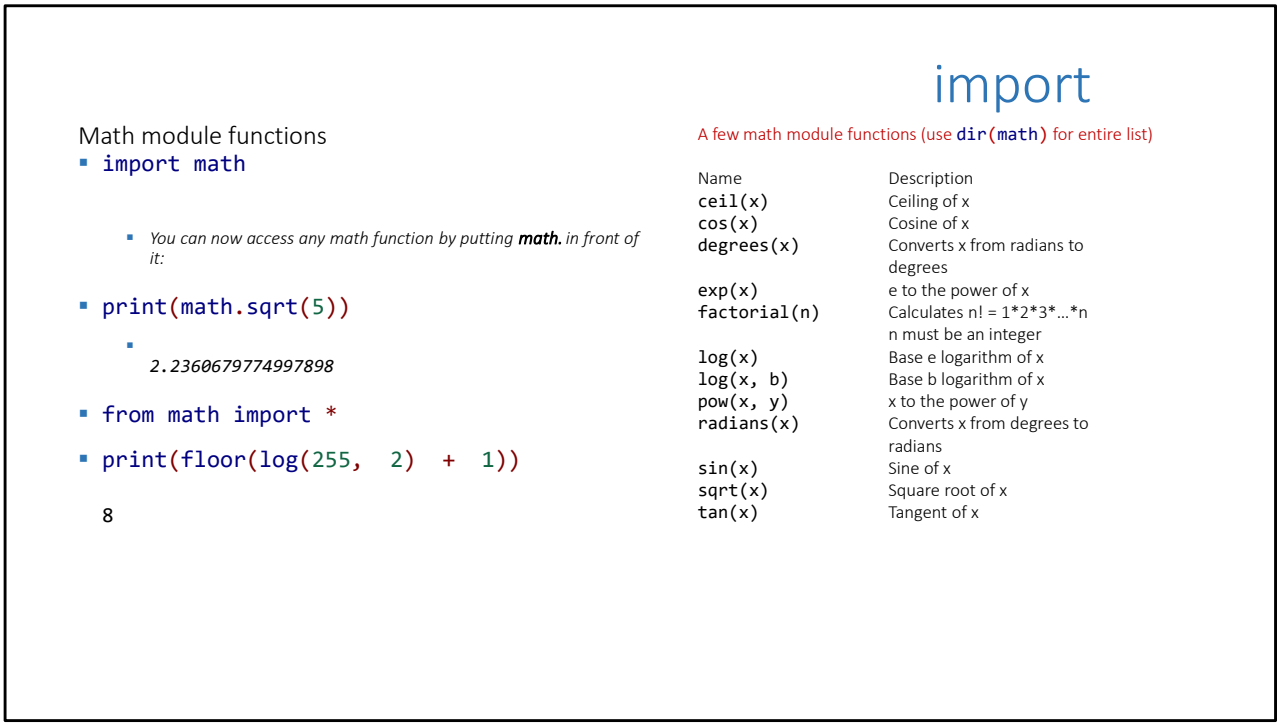
- Write the beginnings of a guessing game in which the program gives hints to the user. 'Pick a number between say, 1 and 10. Count how many guesses are required.



$3n + 1$ Problem (using a set)

```
n = int(input('Enter a whole integer value'))

s = set() # using a set.

while True:

    print(n)

    if n in s: # set membership test.

        break

    s.add(n) # here only if n not a member of set s.

    if n & 1: # n is Odd. Could also use if n % 2 == 1:

        n *= 3
        n += 1

    else: # n is Even.

        n //= 2
```

Set: an unordered collection of items, with no duplicates allowed.



If you want to know where a module *lives*, import it, and then use `help()` like this:

```
import random
help(random)
```

At the bottom of the output you'll see something like:

```
FILE
       c:\program files\python310\lib\random.py
```

However, some modules are built in to the interpreter (over 60 currently). The `math` module is like this. If you try the trick above on that you'll see:

```
FILE
    (built-in)
```

If you want a full list of the available modules you have installed, use `help('modules').` If you want to know which are 'built in', `import sys`, then use `print(sys.builtin_module_names)`. Lastly, if you want to know where Python looks for non built-in modules, use `import sys`, then `print(sys.path)`

Functions

```
import math

n = 100

result = math.factorial(n)
```

```
def factorial(n):

    prod = 1

    for i in range(2, n + 1):

        prod *= i # same as prod = prod * i

    return prod


n = 100

result = factorial(n)
```

A function is like a mini program: processing input and outputting a result.

So, they usually take one or more inputs, and usually return one or more results – but they don't have to do either of those things if they don't want to.

```
def func(a, b):
    return divmod(a, b) # return a // b and also a % b.

print(func(10, 3))
```

```
>>> func(10, 3)
(3, 1)
```

The arguments (that's the inputs) can take default values. `def factorial(n = 10)`, would allow us to call our function with empty parentheses, `print(factorial())`; with `n` defaulting to having the value `10`.

You can read a bit more about argument options here:

https://www.programiz.com/python-programming/function-argument

Functions

```
import math

n = 100

result = math.factorial(n)
```

```
def factorial(n):

    if n == 1:

        return 1

    else:

        return n * factorial(n - 1)


n = 100

result = factorial(n)
```

This is the third and final way to repeat things. It's called *recursion*.

Here we can see that the `factorial(n)` function can, under certain circumstances, call itself. In fact, it will always do so unless its input `n` has the value `1`.

Recursion vs Iteration

```
def fibonacci(n):
    if n < 2:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

n = 11
print([fibonacci(i) for i in range(n)])
```

```
def fibonacci(n):
    a = 1
    b = 1
    for _ in range(1, n):
        t = a
        a = b
        b = t + b
    return a

n = 11
print([fibonacci(n) for n in range(n)])
```

```
>>> 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

One has to be careful with recursion as it's easy to write very elegant, very slow code!

Here, the code on the left is very inefficient – try altering it by increasing values for `n`. Go in small increments of `10`. When it slows down, compare it to the code on the right (speed).

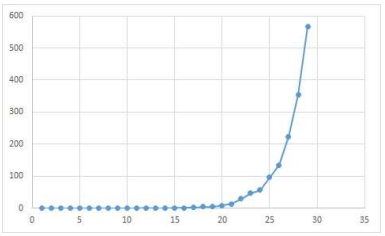
The code on the left can be made to run just as fast as the code on the right by using *caching* - to cache previously computed Fibonacci numbers (this is called *memorization*).

Google for `lru_cache()`, and also have a look at `@functools.cache` here:

https://docs.python.org/3/library/functools.html

Recursion - fibonacci

```
import time
def fibonacci(n):
    if n < 2:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

for n in [1, 5, 10, 15, 20, 25, 30, 35]: # Increase in small increments.
    t1 = time.perf_counter_ns() # Nano second timer.
    k  = fibonacci(n)
    t2 = time.perf_counter_ns()
    print(f'Computing the {n}th fib number ({k}) took {t2 - t1:,} nano seconds')
```



$O(2^n)$

In the code above we're using the library's `time` module to determine something about our code's efficiency.

Copy, paste, and run the code.

You'll see that this doesn't 'scale' well with values of `n`. To fix this, either cache previously seen `n` values using a `dict()`, or look-up *dynamic programming* and `from functools import lru_cache`, and the *decorator* `@lru_cache`.

A bit more *pythonic*

```
def fibonacci(n):
    a, b = 1, 1
    for _ in range(1, n):
        a, b = b, a + b
    return a
print(fibonacci(10))
```

Pythonic means to use the special features of the Python language – rather than using the usual way things might be done in another language, and then translating that into Python code.

`a, b = b, a + b` might look a little unsafe, or at least a little *suspect*?

`a` is assigned the value of b, as in (`a, _ = b, _`), and `b` is assigned to `a + b`, as in (`_, b = _, a + b`), but hold on, didn't we just change `a,` to `b`'s value? So, in assigning `a + b` to `b`, what value of `a` is being used – the one assigned to `a`, by `a, _ = b, _`, or `a`'s original value?

What actually happens here is that the right hand side is resolved first, and then the relevant values are assigned in one step.

https://docs.python.org/3/reference/expressions.html#evaluation-order

Monty Hall

```
import random

doors = [1, 2, 3]

def playGame():

  carDoor    = random.choice(doors)
  playerDoor = random.choice(doors)

  # The host opens a different door to reveal a goat
  # (always able to do this as there are 2 goats).
  hostDoor = random.choice(list(set(doors) - set([carDoor, playerDoor])))

  # ~~~~ To stick, just comment out the next line -
  # it implements that the player is swapping doors.
  playerDoor = (set(doors) - set([playerDoor, hostDoor])).pop()

  return True if playerDoor == carDoor else False
```

```
carsWon = 0

for games in range(1000):
  if playGame():
    carsWon += 1

print('The player won the car ' +
   str(round(carsWon / (games / 100))) +
   '% percent of the time')
```

Using a function to play separate games of the Monty Hall problem.

Strings – immutable, ordered, indexed

```
s = 'hello world'            # variable called s
print(len(s))                # number of characters in s. 11
s = s.title()                # title case s, assign back to s
print(s)                     # 'Hello World'
print(s.find('o'))           # o is 4th char from left starting from 0
print(s.rfind('o'))          # different o in position 7
print(''.join(reversed(s)))  # 'dlroW olleH'
print('Wo' in s)             # True
n = s.find('World')          # n is 6
print(s[n:])                 # 'World'
print(s[0::2])               # 'HloWrd'
print(s[::-1])               # 'dlroW olleH'
```

Strings and 'Slicing'

```
s = 'hello'

# s[start:xstop]
```

	s[0]	s[1]	s[2]	s[3]	s[4]
s =	**h**	**e**	**l**	**l**	**o**

`s[0]` same as `s[0:1]`	**h**	
`s[1]` same as `s[1:2]`		**e**
`s[-4:-2]` or `s[1:3]`	**e**	**l**

The slicing syntax is:

inclusive-starting-position **:** exclusive-ending-position **:** step

If an ending position is negative, it counts from the right hand end. `-1` would give an exclusive ending position as the second to last character. E.g.,

`print(s[0:-1])` outputs `'hell'`.

A step of `-1` reverses a string:

`print(s[::-1])` outputs `'olleh'`.

Slicing works on many other data types, not just strings.

See https://www.youtube.com/watch?v=ajrtAuDg3yw for more.

Class

```
import random

class dice():

    def __init__(self, sides = None):

            self.sides = 6 if sides is None else sides

            self.throws = [n for n in range(1, self.sides + 1)]

    def throw(self): # throw is a method of dice.

        return random.choice(self.throws)


d1 = dice(); d2 = dice(20)

print(d1.throw(), d2.throw())
```

We have seen that methods are bits of functionality that are built-in to objects, e.g., the find() method of a string:

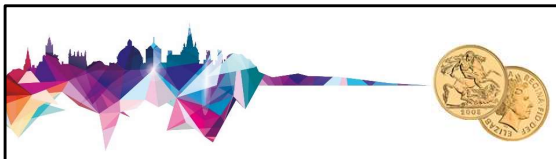
```
s = "Hello World"
print(s.find("World")) # outputs 7.
```

Methods can define the operations that can be performed uniquely on certain object types. E.g., you can only use `find()` on strings, so it makes sense to build it into strings, rather than have it as a global function, like `len()`: `len()` can be used on Tuples; Lists; Dictionaries and Strings.

In Python, we can create our own types, and also the unique operations that may be applied to them.

Here, we are creating a `dice` type, which also contains the definition of what it means to `throw()` a dice instance.

Here, `dice` objects may have any number of `sides`, but have `6` by default. The possible thrown values are also defined as a dice object it is created.

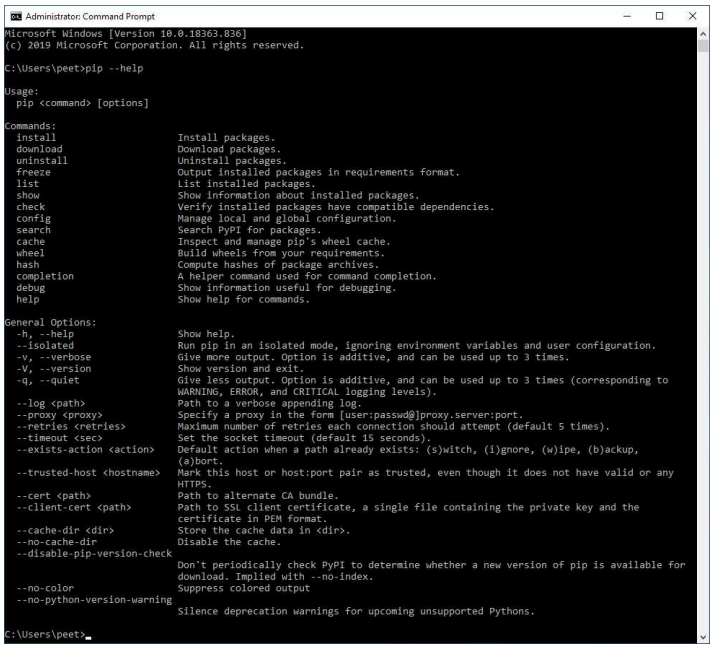


Problem Work-Through

- Write a program to generate a sequence of coin tosses. e.g., HTH.
- On average, how many coin-tosses are needed to see a given sequence?

Sequence	Tosses
HTTTHHTTHHTTTTHTH	17
HHHTTTHHTTHHTH	14
HHHHTH	6
HHTH	4
HHHTTHHHTTTTHHTTHTH	19
HHTTHHHHHTTHHHTTHHHHTH	22
HTTHHHTH	8
TTHHTTTHTTTHTH	14
TTTHHTTTTTHTH	13
TTHTH	5
THTTHTH	7
HTH	3
THHTH	5
HHTTTTTHHHHHTTTHTH	18
HHHTH	5

Average # of tosses to see HTH was …



Package installation (pip/pip3)

- Use pip to
  - *pip install numpy*

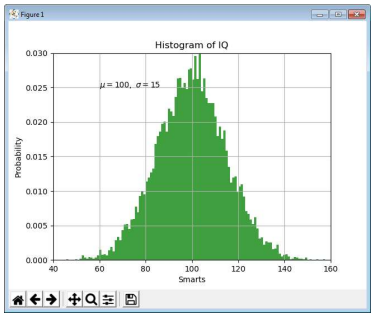
    and

  - *pip install matplotlib*
- Used on the following slide

`pip -list` will show you what packages you have currently installed.

Copy the code below into a new module to test your installation

```
import numpy as np
import matplotlib.pyplot as plt

mu, sigma = 100, 15
data_set = mu + sigma * np.random.randn(10000)

plt.hist(data_set, 125, density = 1, facecolor = 'g', alpha = 0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.text(60, .025, '$\mu=100, \sigma=15$')
plt.xlim(40, 160)
plt.ylim(0, 0.03)
plt.show()
```



Data Science Courses - https://www.linkedin.com/...

Use

`pip –install numpy, matplotlib`

in a command-prompt/terminal before copying/running the code above.

Homework

If HTT beats HTH, what if anything beats HTT? And does something beat that etc? Is there an optimal sequence?

You can use this code to produce a list `l` of all the possible starting permutations:

```
import itertools

rep = 3

l = list(itertools.product('HT', repeat = rep))

for n in range(len(l)):

    l[n] = ''.join(l[n])

print(l, len(l))
```

It turns out that there is no best sequence.

See: https://en.wikipedia.org/wiki/Penney%27s_game

Homework

- Along with the slides from today, and the coin tossing simulation and other code, I am giving you the code of a program that can access and parse RSS feeds, in particular, the BBC's.

- At the end of the code there are some suggestions for modifying the it.

BBC NEWS

RSS Feed For: **BBC News - Technology**

Below is the latest content available from this feed. This isn't the feed I want.

Location data collection firm admits privacy breach
Huq says two apps it collects data from did not seek correct consent from users.

Inside the controversial US gunshot-detection firm
BBC News has been given access to ShotSpotter, the controversial US technology company that detects gunshots.

Facebook: We are putting safety before profits
Facebook Vice President says Meta is about making sure people feel good about using their services.

Meta: Facebook's new name ridiculed by Hebrew speakers
The social media giant joins a number of companies that have fallen foul of translation blunders.

Facebook changes its name to Meta in major rebrand
The social media giant says the new name will better encompass what it does.

UK's electric Arrival promises greener deliveries
A modular design lets vehicles carry the batteries needed for expected trips, avoiding extra weight.

Arm-Nvidia: Europe investigates chip-designer sale
Regulators have "serious doubts" about the technology giant's $40bn takeover of British company Arm.

Chinese payment-terminal company searched by FBI
Pax Technology is a major provider of payment terminals worldwide.

Facebook changes its name to Meta in major rebrand
The social media giant says the new name will better encompass what it does.

UK's electric Arrival promises greener deliveries
A modular design lets vehicles carry the batteries needed for expected trips, avoiding extra weight.

Arm-Nvidia: Europe investigates chip-designer sale
Regulators have "serious doubts" about the technology giant's $40bn takeover of British company Arm.

Chinese payment-terminal company searched by FBI
Pax Technology is a major provider of payment terminals worldwide.

Resources

- *Real Python – Link*
- *Python Programming Tutorials – Link*
- *Linked-In Learning (of course!)*
- *Socratica Python - Link*
- *Anything Python by Corey Schafer - Link*
- *Any YouTube videos by Raymond Hettinger*
- *Pandas (Python Data Analysis Library) - Link*
- *A little more technical: Python as C++'s Limiting Case - Link*

Resources – Me!

If you would like to book a one-to-one session with me, please just email me to arrange a suitable date and time.

Always happy to help. And it's free.

Any final questions?