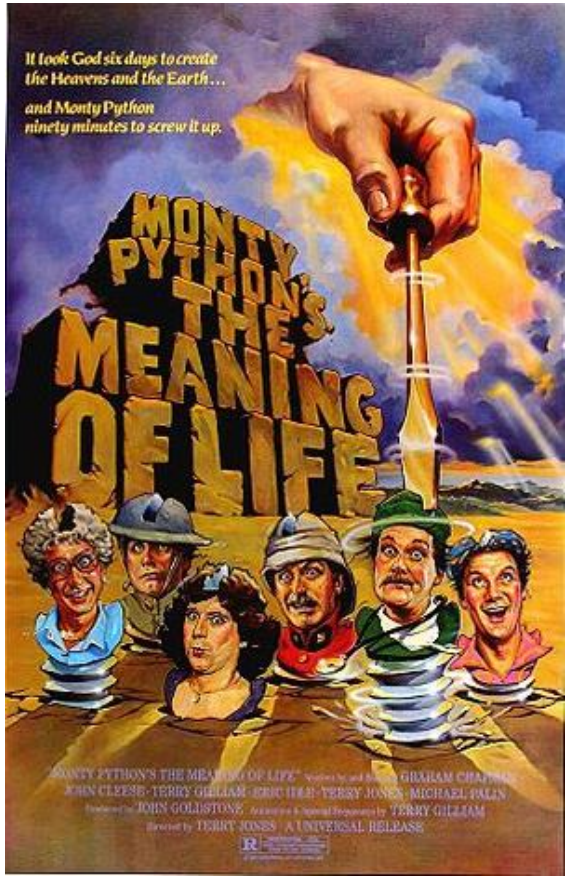
Python Basics

- Functions
- Loops
- Recursion



Built-in functions

```
 >>> type (32)
<type 'int'>
 >>> int('32')
32
```

- From math

```
>>>import math
>>> degrees = 45
>>> radians = degrees/360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071106781187
```

- Composition

```
>>>x = math.exp(math.log(x+1))
```

Adding functions

- Specify the name of a new function and the sequence of statements

```
def print_lyrics():
    print ('Twinkle, twinkle, little star,\n')
    print ('How I wonder what you are?\n')
```

- Note the header ends with a colon and the body is indented.
  - Convention is to indent 4 spaces btw.

- If you're in interactive mode, the interpreter prints ellipses to let you know you aren't done. To end, enter an empty line.

More on functions

- Defining a function creates a variable with same name

```
>>> print print_lyrics
<function print_lyrics at 0xb7e99e9c>
>>> type (print_lyrics>
<type 'function'>
```

- Syntax for calling them is same as built-in
- Can call them in another function

```
>>> def repeat_lyrics():
>>>     print_lyrics()
>>>     print_lyrics()
```

Flow of execution

- You can define a function in the middle of code
  - Just make sure you define a function ***before*** you use it
  - You might do this if you are writing scripts
  - Can do it in code you save – DON'T!!!

Parameters and Arguments

- Arguments are in call statements
- Parameters are in function headings
- Pass arguments to parameters.
  - Pass values to parameters
  - Pass variables to parameters

Example

```
def print_twice (s):
    print (s)
    print (s)

print_twice(17)
print_twice ('Spam')
name = 'Bruce'
print_twice (name)
```

Outputs: `17 17 Spam Spam Bruce Bruce`

- See anything interesting here?

Variables and parameters

- Variables and parameters are local
  - Only exist inside function
  - Do not get a type, until you pass an argument to it
  - Be CAREFUL!!!
    - Implicit variable declarations can cause problems - violates security principle.
- Arguments are **passed** neither by value nor by **reference** in **Python**- instead they are **passed** by assignment. The parameter **passed** in is actually a **reference** to an object, as opposed to **reference** to a fixed memory location but the **reference** is **passed** by value.
  - Immutable objects like integers, strings: passing acts like pass-by-value
  - Mutable objects: pass object reference (by value), so object can be changed

Fruitful and void functions

- When a function yields results – it is called a fruitful function
  - If it doesn't, it is a void function
- Always do something with the result of fruitful function
  - Assign it to a variable
  - Use it in a condition
- if you try to assign result of a void function to a variable, you get a value called None

```
>>> result = print_twice(17)
17  17
>>> print result
None          #note: not 'None'
```

Importing with `from`

- Can import two ways

  `import math` or `from math import pi`

- Latter will allow you to refer to pi directly without dot notation

```
>>>print pi
3.14159265359
```

- Can use star operator to import *everything* from the module

  `from math import *`

  - More concise code (advantage)
  - Naming conflicts (disadvantage)

Recursion

- It is a legal for a function to call itself

```
def countdown(n):
    if n <= 0:
        print ('Blastoff!')
    else:
        print (n)
        countdown(n-1)

countdown(10)
```

range() function

- range is a function to iterate over a sequence of numbers
- `range(n)` generates an iterator to progress from `0` to `n-1`
- `range(begin, end)` generates an iterator to progress from `begin` to `end-1`
- `range(begin, end, step)` generates an iterator to progress from `begin` to `end-1`, incrementing (or decrementing by the step)

```
>>>range (4, 10)
range(4,10)
>>>list(range(4,10))
[4,5,6,7,8,9]
```

Simple Repetition

- For loop

```
for i in range(4):
    print ('Hello!')
```

- while loop

```
n = 10
while n > 0:
    print ('Hello!')
    n = n - 1
print ('Blastoff!')
```

Some questionable loop examples

- Using a break

```
while True:
    line = input ('> ')
    if line == 'done':
       break
    print (line)

print ('Done!')
```

Refactoring loop

- Not using a break

```
line = input ('> ')
while line != 'done':
    print (line)
    line = input ('> ')

print ('Done!')
```

Loop example

```
fibonacci = [0,1,1,2,3,5,8,13,21]
for i in range(len(fibonacci)):
    print(i,fibonacci[i])
print()
```

- Output looks like this :

```
0 0
1 1
2 1
3 2
4 3
5 5
6 8
7 13
8 21
```

For-each loop with optional loop else

```
edibles = ["ham", "spam","eggs","nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)

#loop else, only executed if loop not broken
else:
    print("I am so glad: No spam!")

print("Finally, I finished stuffing myself")
```

Demo – with spam in list and without