

Introduction to Python

Python Conceptual Hierarchy

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

Data Types

Object type	Example literals/creation
Numbers	<code>1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()</code>
Strings	<code>'spam', "Bob's", b'a\x01c', u'sp\xc4m'</code>
Lists	<code>[1, [2, 'three'], 4.5], list(range(10))</code>
Dictionaries	<code>{'food': 'spam', 'taste': 'yum'}, dict(hours=10)</code>
Tuples	<code>(1, 'spam', 4, 'U'), tuple('spam'), namedtuple</code>
Files	<code>open('eggs.txt'), open(r'C:\ham.bin', 'wb')</code>
Sets	<code>set('abc'), {'a', 'b', 'c'}</code>
Other core types	<code>Booleans, types, None</code>

Operators

Arithmetic Operations

Operator	Name	Description
<code>a + b</code>	Addition	Sum of a and b
<code>a - b</code>	Subtraction	Difference of a and b
<code>a * b</code>	Multiplication	Product of a and b
<code>a / b</code>	True division	Quotient of a and b
<code>a // b</code>	Floor division	Quotient of a and b, removing fractional parts
<code>a % b</code>	Modulus	Remainder after division of a by b
<code>a ** b</code>	Exponentiation	a raised to the power of b
<code>-a</code>	Negation	The negative of a
<code>+a</code>	Unary plus	a unchanged (rarely used)

Boolean Operators

Operation	Result
<code>x or y</code>	if x is false, then y, else x
<code>x and y</code>	if x is false, then x, else y
<code>not x</code>	if x is false, then True, else False

Comparison Operations

Operation	Description
<code>a == b</code>	a equal to b
<code>a != b</code>	a not equal to b
<code>a < b</code>	a less than b
<code>a > b</code>	a greater than b
<code>a <= b</code>	a less than or equal to b
<code>a >= b</code>	a greater than or equal to b

Bitwise Operations

Operator	Name	Description
<code>a & b</code>	Bitwise AND	Bits defined in both a and b
<code>a b</code>	Bitwise OR	Bits defined in a or b or both
<code>a ^ b</code>	Bitwise XOR	Bits defined in a or b but not both
<code>a << b</code>	Bit shift left	Shift bits of a left by b units
<code>a >> b</code>	Bit shift right	Shift bits of a right by b units
<code>~a</code>	Bitwise NOT	Bitwise negation of a

Identity and Membership Operators

Operator	Description
<code>a is b</code>	True if a and b are identical objects
<code>a is not b</code>	True if a and b are not identical objects
<code>a in b</code>	True if a is a member of b
<code>a not in b</code>	True if a is not a member of b

Expressions

- A combination of **objects** and **operators** that **computes a value** when executed
 - For instance:
to add two numbers X and Y you would say $X + Y$,

- Rules:

Mixed operators follow operator precedence

Parentheses group subexpressions

Mixed types are converted up

Operator overloading and polymorphism

Expressions

Operators	Description
<code>yield x</code>	Generator function send protocol
<code>lambda args: expression</code>	Anonymous function generation
<code>x if y else z</code>	Ternary selection (x is evaluated only if y is true)
<code>x or y</code>	Logical OR (y is evaluated only if x is false)
<code>x and y</code>	Logical AND (y is evaluated only if x is true)
<code>not x</code>	Logical negation
<code>x in y, x not in y</code>	Membership (iterables, sets)
<code>x is y, x is not y</code>	Object identity tests
<code>x < y, x <= y, x > y, x >= y</code>	Magnitude comparison, set subset and superset;
<code>x == y, x != y</code>	Value equality operators
<code>x y</code>	Bitwise OR, set union
<code>x ^ y</code>	Bitwise XOR, set symmetric difference
<code>x & y</code>	Bitwise AND, set intersection
<code>x << y, x >> y</code>	Shift x left or right by y bits

<code>x + y</code>	Addition, concatenation;
<code>x - y</code>	Subtraction, set difference
<code>x * y</code>	Multiplication, repetition;
<code>x % y</code>	Remainder, format;
<code>x / y, x // y</code>	Division: true and floor
<code>-x, +x</code>	Negation, identity
<code>~x</code>	Bitwise NOT (inversion)
<code>x ** y</code>	Power (exponentiation)
<code>x[i]</code>	Indexing (sequence, mapping, others)
<code>x[i:j:k]</code>	Slicing
<code>x(...)</code>	Call (function, method, class, other callable)
<code>x.attr</code>	Attribute reference
<code>(...)</code>	Tuple, expression, generator expression
<code>[...]</code>	List, list comprehension
<code>{...}</code>	Dictionary, set, set and dictionary comprehensions

Statements

- **Expressions** process **objects** and are embedded in **statements**.
- **Statements** code the larger logic of a program's operation
 - they use and direct expressions to process the objects
- Statements are where objects spring into existence
 - e.g., in expressions within assignment statements
 $X = Y + Z$

Statements

Statement	Role	Example			
Assignment	Creating references	<code>a, b = 'good', 'bad'</code>	<code>def</code>	Functions and methods	<code>def f(a, b, c=1, *d): print(a+b+c+d[0])</code>
Calls and other expressions	Running functions	<code>log.write("spam, ham")</code>	<code>return</code>	Functions results	<code>def f(a, b, c=1, *d): return a+b+c+d[0]</code>
<code>print</code> calls	Printing objects	<code>print('The Killer', joke)</code>	<code>yield</code>	Generator functions	<code>def gen(n): for i in n: yield i*2</code>
<code>if/elif/else</code>	Selecting actions	<code>if "python" in text: print(text)</code>	<code>global</code>	Namespaces	<code>x = 'old' def function(): global x, y; x = 'new'</code>
<code>for/else</code>	Iteration	<code>for x in mylist: print(x)</code>	<code>nonlocal</code>	Namespaces (3.X)	<code>def outer(): x = 'old' def function(): nonlocal x; x = 'new'</code>
<code>while/else</code>	General loops	<code>while X > Y: print('hello')</code>	<code>import</code>	Module access	<code>import sys</code>
<code>pass</code>	Empty placeholder	<code>while True: pass</code>	<code>from</code>	Attribute access	<code>from sys import stdin</code>
<code>break</code>	Loop exit	<code>while True: if exittest(): break</code>	<code>class</code>	Building objects	<code>class Subclass(Superclass): staticData = [] def method(self): pass</code>
<code>continue</code>	Loop continue	<code>while True: if skiptest(): continue</code>	<code>try/except/finally</code>	Catching exceptions	<code>try: action() except: print('action error')</code>
			<code>raise</code>	Triggering exceptions	<code>raise EndSearch(location)</code>
			<code>assert</code>	Debugging checks	<code>assert X > Y, 'X too small'</code>
			<code>with/as</code>	Context managers (3.X, 2.6+)	<code>with open('data') as myfile: process(myfile)</code>
			<code>del</code>	Deleting references	<code>del data[k] del data[i:j] del obj.attr del variable</code>

Statements

Operation	Interpretation
<code>spam = 'Spam'</code>	Basic form
<code>spam, ham = 'yum', 'YUM'</code>	Tuple assignment (positional)
<code>[spam, ham] = ['yum', 'YUM']</code>	List assignment (positional)
<code>a, b, c, d = 'spam'</code>	Sequence assignment, generalized
<code>a, *b = 'spam'</code>	Extended sequence unpacking (Python 3.X)
<code>spam = ham = 'lunch'</code>	Multiple-target assignment
<code>spams += 42</code>	Augmented assignment (equivalent to <code>spams = spams + 42</code>)

Control Flow

The **if**, **while** and **for** statements implement traditional control flow constructs.

Control flow is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated.

The emphasis on explicit control flow distinguishes an *imperative programming* language from a *declarative programming* language.

Within an imperative programming language, a *control flow statement* is a statement which execution results in a choice being made as to which of two or more paths to follow.

Programming - Sieve of Eratosthenes

The **sieve of Eratosthenes** is a simple, ancient algorithm for finding all prime numbers up to any given limit.

1. Create a list of consecutive integers from 2 through n : (2, 3, 4, ..., n).
2. Initially, let p equal 2, the smallest prime number.
3. Enumerate the multiples of p by counting to n from $2p$ in increments of p , and mark them in the list (these will be $2p$, $3p$, $4p$, ...; the p itself should not be marked).
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.
5. When the algorithm terminates, the numbers remaining not marked in the list are all the primes below n .

Task: Write a script that implements the Sieve of Eratosthenes.

Programming - Sieve of Eratosthenes

```
# N - We will find all primes less than N
N = 100

# Create three empty lists
# - One to collect primes
# - One to collect the numbers we haven't checked yet
# - One to collect the composites
primes = []
numbers = []
composites = []

# Initialize the numbers list
for i in range(2,N+1):
    numbers.append(i)

# Loop while there are unchecked numbers
while len(numbers) > 0:

    # The first number in the numbers list is prime
    primes.append(numbers[0])

    # Any multiple of that first number is a composite
    for i in range(numbers[0],N+1,numbers[0]):
        composites.append(i)

    # Use sets to remove the composites
    numbers = list(set(numbers)-set(composites))

    # Sets are unordered, so numbers must be put back in order
    numbers.sort()

# Print the prime list
print(primes)
```

Programming - Simple Primality Testing

One of the simplest primality test is *trial division*:

Given an input number n , check whether any integer m from 2 to \sqrt{n} evenly divides n (the division leaves no remainder).

If n is divisible by any m then n is composite, otherwise it is prime.

Task: Write a script that asks the user for a number and then uses trial division to test if that number is prime.

Programming - Simple Primality Testing

```
# Take some input, convert it to an int
num = int(input('[prime?]>>> '))

# Loop from 2 to the squareroot of the input
for i in range(2, int(num**.5)+1):

    # If any of these numbers divide the input
    # then it is composite and we are done
    if num % i == 0:
        print('Composite')
        quit()

# If we make it through those numbers
# and haven't found a factor
# then the input is prime
print('Prime')
```

File I/O

Before you can read or write a file, you have to open it using Python's built-in `open()` function.
`open(filename, mode)`.

This function creates a **file object**, which would be utilized to call methods associated with it.

It is most commonly used with two arguments.

- ❖ *filename* is a path-like object giving the pathname of the file to be opened.
- ❖ *mode* is an optional string that specifies the mode in which the file is opened. It defaults to `'r'`.

Character	Meaning
<code>'r'</code>	open for reading (default)
<code>'w'</code>	open for writing, truncating the file first
<code>'x'</code>	open for exclusive creation, failing if the file already exists
<code>'a'</code>	open for writing, appending to the end of the file if it exists
<code>'b'</code>	binary mode
<code>'t'</code>	text mode (default)
<code>'+'</code>	open a disk file for updating (reading and writing)
<code>'U'</code>	universal newlines mode (deprecated)

File I/O

To read a file's contents call:

- `fileobject.read(size)`
which reads the specified quantity of data and returns it as a string .
 - `size` is an optional numeric argument. When `size` is omitted or negative, the entire contents of the file will be read and returned.
- `fileobject.readline()`
reads a single line from the file.
- For reading lines from a file, you can loop over the file object.
 - This is memory efficient, fast, and leads to simple code:

```
>>> for line in fileobject:  
...     print(line)  
...
```

```
This is the first line of the file.  
Second line of the file
```


File I/O

- `fileobject.seek(offset, reference point)`
moves to the position that is computed from adding *offset* to a *reference point*.
 - A *reference point* of 0 measures from the beginning of the file
 - 1 uses the current file position
 - 2 uses the end of the file as the reference point
 - *reference point* can be omitted and defaults to 0.
- `fileobject.write(string)`
writes the contents of ***string*** to the file, returning the number of characters written.

File I/O

- When you want to save more complex data types like nested lists and dictionaries Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation).
- The standard module called `json` can take Python data hierarchies, and convert them to string representations. This process is called *serializing*.
- Reconstructing the data from the string representation is called *deserializing*.

File I/O

- If you have an object `x`, you can view its JSON string representation with a simple line of code:

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

- Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a text file. So if `f` is a text file object opened for writing, we can do this:

```
json.dump(x, f)
```

- To decode the object again, if `f` is a text file object which has been opened for reading:

```
x = json.load(f)
```

- This simple serialization technique can handle lists and dictionaries and allows us to store these structures for later use.

File I/O

- `f.close()`

closes the file and frees up any system resources used by it.

- If you don't explicitly close a file, Python's garbage collector will eventually destroy the object and close the open file for you, but the file may stay open for a while.

- It is good practice to use the **with** keyword when dealing with file objects.

- The advantage is that the file is properly closed after its block finishes.

```
>>> with open('workfile') as f:
...     read_data = f.read()
>>> f.closed
True
```

Functions

Programming languages support decomposing problems in several different ways:

- Most programming languages are **procedural**:
 - programs are lists of instructions that tell the computer what to do with the program's input.
 - C, Pascal, and even Unix shells are procedural languages.
- In **declarative** languages:
 - You write a specification that describes the problem to be solved, and the language implementation figures out how to perform the computation efficiently.
 - SQL is the declarative language.
- **Functional** programming:
 - Decomposes a problem into a set of functions.
 - In a functional program, input flows through a set of functions.
 - Each function operates on its input and produces some output.

Functions

Python gives you many built-in functions but you can also create your own functions. These functions are called *user-defined functions*.

A function is a block of organized, reusable code. They group a set of statements so they can be run more than once in a program

Functions provide better modularity for your application and a high degree of code reusing.

Functions also provide a tool for splitting systems into pieces that have well-defined roles.

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

Functions

def Statements

- The def statement creates a function object and assigns it a name.
Its general form:

```
def name(arg1, arg2, ... argN):  
    statements
```

- Function header begins with the keyword **def** followed by the **function name** and **parentheses ()**.
- Any **input parameters** or **arguments** should be placed within these parentheses.
- The code block within every function starts with a **colon :** and is **indented**.

Functions

def Statements

- **def** is an executable statement - it creates a new function object at runtime (as opposed to a compiled language)
- Like all other statements **def** statements can appear where statements are valid.
 - Nested function declarations are allowed

```
if test:
    def func():           # Define func this way
        ...
else:
    def func():           # Or else this way
        ...
...
func()                   # Call the version selected and built
```

```
othername = func          # Assign function object
othername()               # Call func again
```


Functions

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

The keyword **def** introduces a **function definition**.

When Python reaches and runs this **def**, it creates a new function object that packages the function's code and assigns the object to the name **fib2**.

The function definition **DOES NOT** execute the function body.

To execute a function it must be **called**.

- You cannot call a function until after you have defined it.

You can **call** (run) the function in your program by adding parentheses after the function's name.

- The parentheses may optionally contain one or more object arguments

Functions

return leaves the current function call with the expression list.

Even functions without a **return** statement do return a value, `None`.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring; it's good practice to include docstrings in code that you write, so make a habit of it.

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100               # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Functions

argument

A value passed to a function when calling the function. There are two kinds of argument:

- **keyword argument:** an argument preceded by an **identifier** (e.g. `name=`) in a function call or be passed as values in a **dictionary preceded by `**`**.
 - For example, `3` and `5` are both keyword arguments in the following calls to `complex()`:
`complex(real=3, imag=5)`
- **positional argument:** an argument that is not a keyword argument.
Positional arguments can appear at the **beginning** of an argument list or be passed as elements of an **iterable preceded by `*`**.
 - For example, `3` and `5` are both positional arguments in the following calls:
`complex(3, 5)`

Functions

Default Argument Values

```
def ask_ok(prompt, retries=4, reminder='Please try again!):
```

Keyword Arguments

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue):
```

Arbitrary Argument Lists

```
def write_multiple_items(file, separator, *args):
```

Functions

Default Argument Values

When one or more parameters have the form *parameter* `=` *expression*, the function is said to have “default parameter values.”

For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter’s default value is substituted.

If a parameter has a default value, all following parameters must also have a default value.

Functions

Default Argument Values

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Functions

Default Argument Values

Default parameter values are evaluated from left to right when the function definition is executed.

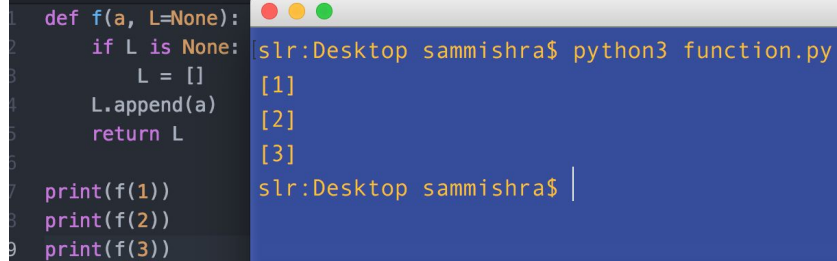
The default value is evaluated only once.

This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes.

```
def f(a, L=[]):  
    L.append(a)  
    return L
```

```
print(f(1))  
print(f(2))  
print(f(3))
```

```
[1]  
[1, 2]  
[1, 2, 3]
```



The screenshot shows a code editor window with a Python script and a terminal window showing the output. The script defines a function `f(a, L=None)` that appends `a` to `L` if `L` is `None`, and then returns `L`. The script calls `print(f(1))`, `print(f(2))`, and `print(f(3))`. The terminal shows the output of these calls: `[1]`, `[2]`, and `[3]`.

```
def f(a, L=None):  
    if L is None:  
        L = []  
    L.append(a)  
    return L  
  
print(f(1))  
print(f(2))  
print(f(3))
```

slr:Desktop sammishra\$ python3 function.py
[1]
[2]
[3]
slr:Desktop sammishra\$

Functions

Keyword Arguments

The following function accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`):

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):  
    print("-- This parrot wouldn't", action, end=' ')  
    print("if you put", voltage, "volts through it.")  
    print("-- Lovely plumage, the", type)  
    print("-- It's", state, "!")
```

This function can be called in any of the following ways:

```
parrot(1000) # 1 positional argument  
parrot(voltage=1000) # 1 keyword argument  
parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments  
parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments  
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments  
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```


Functions

Keyword Arguments

The following function accepts one required argument (voltage) and three optional arguments (state, action, and type):

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):  
    print("-- This parrot wouldn't", action, end=' ')  
    print("if you put", voltage, "volts through it.")  
    print("-- Lovely plumage, the", type)  
    print("-- It's", state, "!")
```

All the following calls would be invalid:

```
parrot()           # required argument missing  
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument  
parrot(110, voltage=220)  # duplicate value for the same argument  
parrot(actor='John Cleese') # unknown keyword argument
```

Functions

Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments.

- Formal parameter of the form `**name` - receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter.
- Formal parameter of the form `*name` - receives a tuple containing the positional arguments beyond the formal parameter list.

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Palindromes

A **palindrome** is a word which reads the same backward as forward, such as *madam* or *racecar*.

Task: Given a word as input determine if it is a palindrome. Return True if the word is a palindrome, False if it is not.

Palindromes

```
def palindrome(forward):  
  
    print(forward)  
    backward = ''.join(reversed(forward))  
    print(backward)  
  
    if backward == forward:  
        return True  
    else:  
        return False  
  
def main():  
    word = 'mommy'  
    print(palindrome(word))  
  
main()
```

Palindromes

```
def palindrome(forward):  
  
    print(forward)  
    backward = forward  
    backward = list(backward)  
    backward.reverse()  
    backward = ''.join(backward)  
    print(backward)  
  
    if backward == forward:  
        return True  
    else:  
        return False  
  
def main():  
    word = 'mommy'  
    print(palindrome(word))  
  
main()
```

Palindromes

```
palindromes.py
1 def get_word():
2     '''Get the word to be checked from user input'''
3
4     word = input("Enter A Word: ")
5     return word
6
7 def check_word(word):
8     '''Check if the word is a palindrome'''
9
10    length = len(word)
11    for i in range(length//2):
12        left = i
13        right = -(i+1)
14        if word[left] != word[right]:
15            return False
16        return True
17
18 def main():
19     print("This script checks if a word is a Palindrome.")
20     word = get_word()
21     print(check_word(word))
22
23 main()
```

Anagrams

An **anagram** is the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once; for example, the word anagram can be rearranged into "nag a ram".

Task: Given two words as input determine if they are an anagram of one-another. Return True if the word is an anagram, False if it is not.

Anagrams

```
9 def check_words(word1, word2):
10     '''Check if two words are anagrams of one-another'''
11
12     if len(word1) != len(word2):
13         return False
14
15     dict1 = {}
16     dict2 = {}
17
18     for i in word1:
19         if i not in dict1:
20             dict1[i] = 0
21         else:
22             dict1[i] += 1
23
24     for i in word2:
25         if i not in dict2:
26             dict2[i] = 0
27         else:
28             dict2[i] += 1
29
30     if dict1 == dict2:
31         return True
32     else:
33         return False
```

```
1 def get_words():
2     '''Get the words to be checked from user input'''
3
4     word1 = input("Enter A Word: ")
5     word2 = input("Enter A Word: ")
6
7     return word1, word2
```

```
def main():
    print('This script checks if two words are anagrams of each other.')
    word1, word2 = get_words()
    print(check_words(word1, word2))

main()
```