**Variable Assignment**

**Rules for variable names**

* names can not start with a number
* names can not contain spaces, use \_ intead
* names can not contain any of these symbols:

:'",<>/?|\!@#%^&\*~-+

* it's considered best practice ([PEP8](https://www.python.org/dev/peps/pep-0008/#function-and-variable-names)) that names are lowercase with underscores
* avoid using Python built-in keywords like list and str
* avoid using the single characters l (lowercase letter el), O (uppercase letter oh) and I (uppercase letter eye) as they can be confused with 1 and 0

**Dynamic Typing**

Python uses *dynamic typing*, meaning you can reassign variables to different data types. This makes Python very flexible in assigning data types; it differs from other languages that are *statically typed*.

my\_dogs **=** 2

​

my\_dogs **=**"Dog"

my\_dogs

print(type(my\_dogs))

<class 'int'>

my\_dogs **=** ['Sammy', 'Frankie']

print(type(my\_dogs))

<class 'list'>

my\_dogs

Out[4]:

['Sammy', 'Frankie']

**Pros and Cons of Dynamic Typing**

**Pros of Dynamic Typing**

* very easy to work with
* faster development time

**Cons of Dynamic Typing**

* may result in unexpected bugs!
* you need to be aware of type()

**Assigning Variables**

Variable assignment follows name = object, where a single equals sign = is an *assignment operator*

In [5]:



a **=** 5

In [6]:



a

Out[6]:

5

Here we assigned the integer object 5 to the variable name a.  
Let's assign a to something else:

In [7]:



a **=** 10

In [8]:



a

Out[8]:

10

You can now use a in place of the number 10:

In [9]:



a **+** a

Out[9]:

20

**Reassigning Variables**

Python lets you reassign variables with a reference to the same object.

In [10]:



a **=** a **+** 10

In [11]:



a

Out[11]:

20

There's actually a shortcut for this. Python lets you add, subtract, multiply and divide numbers with reassignment using +=, -=, \*=, and /=.

In [12]:



​

a **+=** 10

​

a **=** a**+**10

a**+=**10

In [13]:



a

Out[13]:

30

In [14]:



a **\*=** 2

In [15]:



a

Out[15]:

60

**Determining variable type with type()**

You can check what type of object is assigned to a variable using Python's built-in type() function. Common data types include:

* **int** (for integer)
* **float**
* **str** (for string)
* **list**
* **tuple**
* **dict** (for dictionary)
* **set**
* **bool** (for Boolean True/False)

In [16]:



type(a)

Out[16]:

int

In [17]:



a **=** (1,2)

In [18]:



type(a)

Out[18]:

tuple

**Simple Exercise**

This shows how variables make calculations more readable and easier to follow.

In [19]:



my\_income **=** 100

tax\_rate **=** 0.1

my\_taxes **=** my\_income **\*** tax\_rate

In [20]:



my\_taxes

Out[20]:

10.0

Type *Markdown* and LaTeX: 𝛼2α2

In [ ]:



​

In [ ]:



​

In [19]:



'''

To find the value of Y using pythogoras theorem

'''

​

x,y,z **=** 2,3,4

​

function(a, b)

​

​

​

​

​

​

d **=** (2**\*\***2 **+** 3**\*\***2 **+**4**\*\***2)

d**=**(d **\*\*** 0.5)

d

​

Out[19]:

5.385164807134504

Note: Methods do not change the original variable unless we explicitly reassign the variable. So, the name has not been changed, despite the methods that we have applied.

Type *Markdown* and LaTeX: 𝛼2α2

**Annotated assignment statements**

Type Annotations are a new feature added in PEP 484 that allow for adding type hints to variables. They are used to inform someone reading the code what the type of a variable should be. This brings a sense of statically typed control to the dynamically typed Python. This is accomplished by adding : after initializing/declaring a variable. An example is shown below, which adds the : int when we declare the variable to show that age should be of type int.

In [1]:



add :int**=** 5

add :int**=** 5

​

add: int **=**5

print(add)

​

sub:float **=** 4.5

​

5

It is important to note that type annotations do not affect the runtime of the program in any way. These hints are ignored by the interpreter and are used solely to increase the readability for other programmers and yourself. They are not enforced are runtime A helpful feature of statically typed languages is that the value of a variable can always be known within a specific domain. For instance, we know string variables can only be strings, ints can only be ints, and so on. With dynamically typed languages, its basically anyone's guess as to what the value of a variable is or should be.

We can use the expected variable's type when writing and calling functions to ensure we are passing and using parameters correctly. If we pass a str when the function expects an int, then it most likely will not work in the way we expected.

In [8]:



**def** mystery(a,b,times):

**return**(a**+**b)**\***times

​

*#We can see what that function is doing, but do we know what a, b, or times are supposed to be*

In [12]:



print(mystery(2,3,5))

​

print(mystery('first','second',4))

​

print("\*"**\***10)

​

​

25

firstsecondfirstsecondfirstsecondfirstsecond

\*\*\*\*\*\*\*\*\*\*

based on what we pass the function, we get two totally different results. With integers we get some nice PEMDAS math, but when we pass strings to the function, we can see that the first two arguments are concatenated, and that resulting string is multiplied times times.

In [7]:



**def** mystery(a:str,b:str,times:int)**->**str: *#return value from function is str which mentioned using ->str*

**return**(a**+**b)**\***times

​

We have added : str, : str, and : int to the function's parameters to show what types they should be. This will hopefully make the code clearer to read, and reveal it's purpose a little more.

We also added the -> str to show that this function will return a str. Using -> , we can more easily show the return value types of any function or method, to avoid confusion by future developers!

In [5]:



print(mystery(2,3,5))

​

print(mystery('first','second',4))

25

firstsecondfirstsecondfirstsecondfirstsecond

Again, we can still call our code in the first, incorrect way, but hopefully with a good review, a programmer will see that they are using the function in a way it was not intended. Type annotations and hints are incredibly useful for teams and multi-developer Python applications. It removes most of the guesswork from reading code!

In [6]:



*#To check what are all the attributes \_\_annotations\_\_ has we can use dir function*

print(dir(\_\_annotations\_\_))

['\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']

In [1]:



print(help('mystery'))

print(dir('mystery'))

No Python documentation found for 'mystery'.

Use help() to get the interactive help utility.

Use help(str) for help on the str class.

None

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

What we have here is a Python file that we have named annotate.py. In it, we have created a variable, name, and annotated it to indicate it is a string. This is done by adding a colon after the variable name and then specifying what type it should be. You don’t have to assign anything to the variable if you don’t want to. The following is just as valid:

In [2]:



**import** annotate *#annotate.py*

print((annotate.\_\_annotations\_\_)) *#to check what are the annotations used in annotate file/program there is int and str values*

​

​

​

{'name': <class 'str'>, 'age': <class 'int'>}

In [5]:



**class** Mymodule:

firstname:str *#is valid without defining alos while specifying the datatyoe*

firstname1 **=**'abc' *#just declaration is error , with value it will become class variable*

**def** \_\_init\_\_(self,x : str):

self.firstname **=**x

**def** display(self):

print(self.firstname)

Obj **=** Mymodule('Mike')

Obj.display()

Mike

In [6]:



print(Mymodule.\_\_annotations\_\_) *#you can check the annotations present in your class*

{'firstname': <class 'str'>}

**Complex Types**

For anything more than a primitive in Python, use the typing class. It describes types to type annotate any variable of any type. It comes preloaded with type annotations such as Dict, Tuple, List, Set, and more! Then you can expand your type hints into use cases like the example below.

In [42]:



**from** typing **import** **\*** *#typing class describes types to anotate for dict,set etc*

**def** print\_lnames(lnames : List[str])**->None**: *#this function takes List class with str values and returns none*

**for** student **in** lnames:

print(student)

**def** print\_dictnames(dname :Dict[str,int])**->None**:

**for** names,values **in** dname.items():

print(names ,'- >' ,values)

lnames **=** ['john','mike','alex']

*# lnames =[2,3,4,5] #can call withother datatypes also but readability is lost*

*# lnames ='Mike' # can even call with any other data type also no harm but readability is lost*

print\_lnames(lnames)

dnames **=** {'john':50,'Mike':60,'Alex':70}

print\_dictnames(dnames)

​

john

mike

alex

john - > 50

Mike - > 60

Alex - > 70

In [15]:



**!**python annotate.py

​

Asha

In [17]:



**!**python Testing.py

​

In [18]:



**!**python Testing.py

50

In [ ]:



​

### Values and Data types

A value is one of the basic things a program works with, like a letter or a number. The values we have seen so far are 1, 2, and 'Hello, World!'.

These values belong to different types:

1.) 2 is an integer.

2.)'Hello, World!' is a string, so-called because it contains a ``string'' of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks. The print statement also works for integers

In [1]:



print(4)

​

lst **=**[1,2,3,4]

print(type(lst))

4

<class 'list'>

In [10]:



​

print(type(4))

<class 'int'>

In [3]:



print(type('Hello World'))

<class 'str'>

In [13]:



print(type(3.2))

<class 'float'>

strings belong to the type str and integers belong to the type int. Less obviously, numbers with a decimal point belong to a type called float, because these numbers are represented in a format called floating-point.

What about values like '17' and '3.2'? They look like numbers, but they are in quotation marks like strings They are strings.

In [18]:



print(type('17'))

<class 'str'>

In [16]:



print(type("3.2"))

<class 'str'>

In [17]:



print(type("""3.2"""))

<class 'str'>

In [4]:



a **=**5

print(type(a))

isinstance(a,int)

<class 'int'>

Out[4]:

True

In [5]:



c**=**1**+**2j

print(type(c))

<class 'complex'>

In [ ]:



​

In [ ]:



​

When you type a large integer, you might be tempted to use commas between groups of three digits,as in 1,000,000. This is not a legal integer in Python, but it is illegal:

printing 1,000,000 will provide output as 1 0 0

Well, that's not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers, which it prints with spaces between.

In [3]:



print(100,0000,00)

100 0 0

This is an example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the ``right'' thing.

### Variables

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value.

An assignment statement creates new variables and gives them values:

message = 'And now for something completely different'

n = 17

pi = 3.1415926535897931

This example makes three assignments. The first assigns a string to a new variable named message; the second assigns the integer 17 to n; the third assigns the (approximate) value of p to pi.

To display the value of a variable, you can use a print statement:

In [3]:



message **=**'And now for something completely different'

n**=**17

pi**=**3.145926

print(n)

print(message)

print(pi)

17

And now for something completely different

3.145926

In [ ]:



​

### Variable names and keywords

Programmers generally choose names for their variables that are meaningful---they document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter .

The underscore character (\_) can appear in a name. It is often used in names with multiple words, such as my\_name or airspeed\_of\_unladen\_swallow.

If you give a variable an illegal name, you get a syntax error:

76trombones = 'big parade' SyntaxError: invalid syntax

more@ = 1000000 SyntaxError: invalid syntax

class = 'Advanced Theoretical Zymurgy' SyntaxError: invalid syntax

76trombones is illegal because it does not begin with a letter. more@ is illegal because it contains an illegal character, @. But what's wrong with class?

It turns out that class is one of Python's keywords. The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

Python reserves 31 keywords1 for its use:

and del from not while

as elif global or with

assert else if pass yield

break except import print

class exec in raise

continue finally is return

def for lambda try

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

In [8]:



​

​

In [8]:



76trombones **=**'big parade'

**File "<ipython-input-8-0573dc4ac394>", line 1**

**76trombones ='big parade'**

**^**

**SyntaxError:** invalid syntax

### Statements

A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print and assignment.

When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.

A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

print(1)

x = 2

print(x)

produces the output

1

2

The assignment statement produces no output.

In [27]:



​

print(1)

x **=**2

print(x)

​

print(1)

x**=**2

print(x)

​

​

1

2

In [ ]:



​

## Numbers and more in Python!

we will learn about numbers in Python and how to use them.

We'll learn about the following topics:

1.) Types of Numbers in Python

2.) Basic Arithmetic

3.) Differences between classic division and floor division

4.) Object Assignment in Python

## Types of numbers

Python has various "types" of numbers (numeric literals). We'll mainly focus on integers and floating point numbers.

Integers are just whole numbers, positive or negative. For example: 2 and -2 are examples of integers.

Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (e) to define the number. For example 2.0 and -2.1 are examples of floating point numbers. 4E2 (4 times 10 to the power of 2) is also an example of a floating point number in Python.

Here is a table of the two main types we will spend most of our time working with some examples:

|  |  |
| --- | --- |
| **Examples** | **Number "Type"** |
| 1,2,-5,1000 | Integers |
| 1.2,-0.5,2e2,3E2 | Floating-point numbers |

In [9]:



*#python directly does not support double datatype*

a**=**3

print(1**/**double(a)) *# without import there is error as double is not defined*

**---------------------------------------------------------------------------**

**NameError** Traceback (most recent call last)

**<ipython-input-9-ab1731e02102>** in <module>

1 **#python directly does not support double datatype**

2 a**=3**

**----> 3** print**(1/**double**(**a**))** **# without import there is error as double is not defined**

**NameError**: name 'double' is not defined

In [11]:



*#it is present in one of the module numpy , which needs to be installed and imported and then use*

**from** numpy **import** **\***

a**=** 3

1**/**double(a)

Out[11]:

0.3333333333333333

Now let's start with some basic arithmetic.

Type Markdown and LaTeX: 𝛼2α2

### Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands.

The operators +, -, , / and \* perform addition, subtraction, multiplication, division and exponentiation, as in the following examples:

20+32

hour-1

hour\*60+minute

minute/60

5\*\*2

(5+9)\*(15-7)

The division operator might not do what you expect:

minute = 59

minute/60

0 The value of minute is 59, and in conventional arithmetic 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Python is performing floor division.

When both of the operands are integers, the result is also an integer; floor division chops off the fraction part, so in this example it rounds down to zero.

If either of the operands is a floating-point number, Python performs floating-point division, and the result is a float:

minute/60.0

0.98333333333333328

Type Markdown and LaTeX: 𝛼2α2

In [1]:



print(6**/**3) *#with single slash the out is always fraction irrespective of operands*

​

print(6.0**/**4.0)

2.0

1.5

In [2]:



print(6**//**4) *#with double slash , if it is integer the output is always integer and floor division occurs*

*#and the fractional part is chopped off*

​

print(59**//**60)

1

0

In [1]:



print(2**\***3)

print("@"**\***5)

​

print('#'**\***5)

6

@@@@@

#####

### Expressions

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable x has been assigned a value):

17

x

x + 17

If you type an expression in interactive mode, the interpreter evaluates it and displays the result: 1 + 1

2

# Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. For mathematical operators, Python follows mathematical convention. The acronym PEMDAS is a useful way to remember the rules:

Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 \* (3-1) is 4, and (1+1)\*\*(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute \* 100) / 60, even if it doesn't change the result.

Exponentiation has the next highest precedence, so 2**1+1 is 3, not 4, and 3\*1**3 is 3, not 27.

Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence. So 2\*3-1 is 5, not 4, and 6+4/2 is 8, not 5.

Operators with the same precedence are evaluated from left to right. So in the expression 5-3-1 is 1, not 3 because the 5-3 happens first and then 1 is subtracted from 2. When in doubt always put parentheses in your expressions to make sure the computations are performed in the order you intend.

### Modulus operator

The modulus operator works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

quotient = 7 / 3

print(quotient)

2 remainder = 7 % 3

print(remainder)

1

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another---if x % y is zero, then x is divisible by y.

Also, you can extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly x % 100 yields the last two digits.

In [12]:



*# Modulo*

7**%**4

Out[12]:

3

4 goes into 7 once, with a remainder of 3. The % operator returns the remainder after division.

### Basic Arithmetic

In [13]:



*# Can use parentheses to specify orders*

(2**+**10) **\*** (10**+**3)

Out[13]:

156

In [9]:



*# Order of Operations followed in Python*

2 **+** 10 **\*** 10 **+** 3

In [47]:



*# Can also do roots this way*

4**\*\***0.5

Out[47]:

2.0

In [46]:



*# Powers*

2**\*\***3

Out[46]:

8

In [4]:



*# Addition*

2**+**3

Out[4]:

5

In [40]:



*# Subtraction*

2**-**1

Out[40]:

1

In [41]:



*# Multiplication*

2**\***2

Out[41]:

4

In [3]:



*# Division*

4**/**2.5

Out[3]:

1.6

In [2]:



*# Floor Division*

5**//**2.0

Out[2]:

2.0

In [5]:



8**/**3.0

Out[5]:

2.6666666666666665

In [6]:



8**//**3.0 *#florr division just chops off the precesion values and rounds up , since one of the operrand is floating , the result*

*#also will be float as it will be doing floating point division*

​

Out[6]:

2.0

**So what if we just want the remainder after division?**

In [ ]:



​

The reason we get this result is because we are using "floor" division. The // operator (two forward slashes) truncates the decimal without rounding, and returns an integer result.

## Variable Assignments

We use a single equals sign to assign labels to variables. Let's see a few examples of how we can do this.

In [11]:



*# Let's create an object called "a" and assign it the number 5*

a **=** 5

Type Markdown and LaTeX: 𝛼2α2

In [11]:



*# Adding the objects*

*#a+a*

What happens on reassignment? Will Python let us write it over?

In [12]:



*# Reassignment*

*#a = 10*

In [13]:



*# Check*

*#a*

Python allows you to write over assigned variable names. We can also use the variables themselves when doing the reassignment.

In [14]:



*# Check*

*#a*

In [15]:



*# Use A to redefine A*

*#a = a + a*

In [16]:



*# Check*

*#a*

The names you use when creating these labels need to follow a few rules:

1. Names can not start with a number.

2. There can be no spaces in the name, use \_ instead.

3. Can't use any of these symbols :'",<>/?|\()!@#$%^&\*~-+

4. It's considered best practice (PEP8) that names are lowercase.

5. Avoid using the characters 'l' (lowercase letter el), 'O' (uppercase letter oh),

or 'I' (uppercase letter eye) as single character variable names.

6. Avoid using words that have special meaning in Python like "list" and "str"

Using variable names can be a very useful way to keep track of different variables in Python. For example:

In [4]:



*# Use object names to keep better track of what's going on in your code!*

my\_income **=** 100

​

tax\_rate **=** 0.1

​

my\_taxes **=** my\_income**\***tax\_rate

In [7]:



*# Show my taxes!*

print(my\_taxes)

my\_taxes

10.0

Out[7]:

10.0

# Advanced Numbers

In this lecture we will learn about a few more representations of numbers in Python.

## Hexadecimal

Using the function hex() you can convert numbers into a [hexadecimal](https://en.wikipedia.org/wiki/Hexadecimal) format:

In [1]:



hex(246)

Out[1]:

'0xf6'

In [2]:



hex(512)

Out[2]:

'0x200'

## Binary

Using the function bin() you can convert numbers into their [binary](https://en.wikipedia.org/wiki/Binary_number) format.

In [3]:



bin(1234)

Out[3]:

'0b10011010010'

In [4]:



bin(128)

Out[4]:

'0b10000000'

In [5]:



bin(512)

Out[5]:

'0b1000000000'

## Exponentials

The function pow() takes two arguments, equivalent to x^y. With three arguments it is equivalent to (x^y)%z, but may be more efficient for long integers.

In [1]:



pow(3,4) *#functions*

​

Out[1]:

81

In [7]:



pow(3,4,5)

Out[7]:

1

## Absolute Value

The function abs() returns the absolute value of a number. The argument may be an integer or a floating point number. If the argument is a complex number, its magnitude is returned.

In [8]:



abs(**-**3.14)

Out[8]:

3.14

In [9]:



abs(3)

Out[9]:

3

In [3]:



abs(**-**3**+**2j)

​

*#magnitude here means underroot((3^2)+(2^2)) = underroot(13) = 3.6*

​

Out[3]:

3.6055512754639896

## Round

The function round() will round a number to a given precision in decimal digits (default 0 digits). It does not convert integers to floats.

In [10]:



round(3,2)

Out[10]:

3

In [11]:



round(395,**-**2)

Out[11]:

400

In [12]:



round(3.1415926535,2)

Out[12]:

3.14

Python has a built-in math library that is also useful to play around with in case you are ever in need of some mathematical operations. Explore the documentation (<https://docs.python.org/3/library/math.html>)!