
PYTHON

Introduction to Python Language

Prepared based on the slides from the Computer Programming lecture slides of Wellesley College.

Python Intro Overview

- **Values**: `10` (integer),

  `3.1415` (decimal number or float),

  `'GAZİANTEP'` (text or string)

- **Types**: numbers and text: **`int`**, **`float`**, **`str`**

```
type(10)
type('gaziantep')
```

- **Operators: + - * / % =**

- **Expressions:** (they always produce a value as a result)

```
'abc' + 'def' -> 'abcdef'
```

Knowing the **type** of a **value** allows us to choose the right **operator** when creating **expressions**.

Simple Expressions: Python as calculator

Input Expressions In [...]	Output Values Out [...]	
`1+2`	`3`	
`3*4`	`12`	
`3 * 4`	`12`	`# Spaces don't matter`
`3.4*5.67`	`19.278`	`# Floating point (decimal) operations`
`2 + 3*4`	`14`	`# Precedence: * binds more tightly than +`
`(2+3)*4`	`20`	`# Overriding precedence with parentheses`
`11 / 4`	`2.75` →	`# Floating point (decimal) division`
`11 // 4`	`2` →	`# Integer division`
`11 % 4`	`3` →	`# Remainder (often called modulus)`
`5 - 3.4`	`1.6`	
`3.25 * 4`	`13.0`	
`11.0// 2`	`5.0`	`# output is float if at least one input is float`
`5// 2.25`	`2.0`	
`5 % 2.25`	`0.5`	

Strings and concatenation

A string is just a sequence of characters that we write between a pair of double quotes or a pair of single quotes. Strings are usually displayed with single quotes. **The same string value is created regardless of which quotes are used.**

In [...]	Out [...]	
`"ME444"`	`'ME444'`	# Characters in a string
`'rocks!'`	`'rocks!'`	# can include spaces,
`'You say "Hi!"'`	`'You say "Hi!"'`	# punctuation, quotes
`"No, I didn't"`	`"No, I didn't"`	
`"ME444 " + 'rocks!'`	`'ME444 rocks!'`	# String concatenation
`'123' + '4'`	`'1234'`	# Strings and numbers
`123 + 4`	`127`	# are very different!
`'123' + 4`	`TypeError`	# Can't concatenate strings & num.
`'123' * 4`	`'123123123123'`	# Repeated concatenation
`'123' * '4'`	`TypeError`	

Memory Diagram Model: Variable as a Box

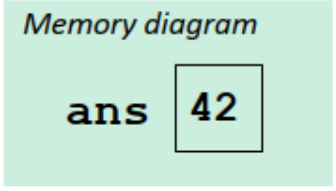
- A variable is a way to remember a value for later in the computer's memory.
- A variable is created by an **assignment statement**, whose form is

varName = expression

Example: ans = 42 # ans is the varName, 42 is the expression saved in ans

This line of code is executed in two steps:

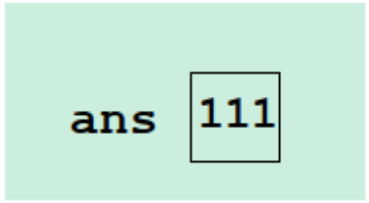
1. Evaluate ***expression*** to its value ***val***
2. If there is no variable box already labeled with ***varName,*** create a new box labeled with ***varName*** and store ***val*** in it; otherwise, change the contents of the existing box labeled ***varName*** to ***val***.



Memory Diagram Model: Variable as a Box

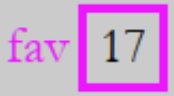
- How does the memory diagram change if we evaluate the following expression?

```
ans = 2*ans+27
```



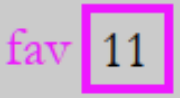
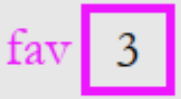
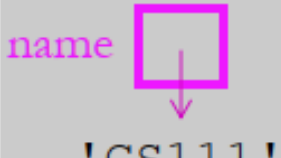
- The expression checks the most recent ***val*** of **ans**(42), re-evaluates the new expression based on that value, and reassigns the value of **ans** accordingly.
  - **ans = 2*42+27**
  - **ans = 111**

Variable Examples

In [...]	Memory Diagram	Out [...]	Notes
`fav = 17`			Assignment statements makes box, no output
`fav`		17	Returns current contents of fav
`fav + fav`		34	The contents of fav are unchanged
`lucky = 8`			Makes new box, has no output
`fav + lucky`		25	Variable contents unchanged
`aSum = fav + lucky`			Makes new box, has no output
`aSum * aSum`		625	Variable contents unchanged

Variable Examples

How does the memory diagram change when we change the values of our existing variables? How are strings stored in memory?

In [...]	Memory Diagram	Out [...]	Notes
`fav = 11`			Change contents of fav box to 11
`fav = fav - lucky`			Change contents of fav box to 3
`name = 'CS111'`			Makes new box containing string. Strings are drawn *outside* box with arrow pointing to them (b/c they're often "too big" to fit inside box
`name*fav`		`'CS111CS111CS111'`	string*int will repeat the string int # of times

Built-in Functions

Built-in function	Result
`max`	Returns the largest item in an iterable (An iterable is an object we can loop over, like a list of numbers. We will learn about them soon!)
`min`	Returns the smallest item in an iterable
`id`	Returns memory address of a value
`type`	Returns the type of a value
`len`	Returns the length of a sequence value (strings are an example)
`str`	Converts and returns the input as a string
`int`	Converts and returns the input as an integer number
`float`	Converts and returns the input as a floating point number
`round`	Rounds a number to nearest integer or decimal point
`print`	Prints a specified message on the screen/output device, and returns the `None` value.
`input`	Asks user for input, converts input to a string, returns the string

Built-in Functions: max and min functions

Python has many built-in functions that we can use. Built-in functions and user-defined variable and function names names are highlighted with different colors in the compiler.

In [...]	Out [...]
`min(7, 3)`	`3`
`max(7, 3)`	`7`
`min(7, 3, 2, 8.19)`	`2 # can take any num. of arguments`
`max(7, 3, 2, 8.19)`	`8.19`
`smallest = min(-5, 2)`	`# smallest gets -5`
`largest = max(-3.4, -10)`	`# largest gets -3.4`
`max(smallest, largest, -1)`	`-1`

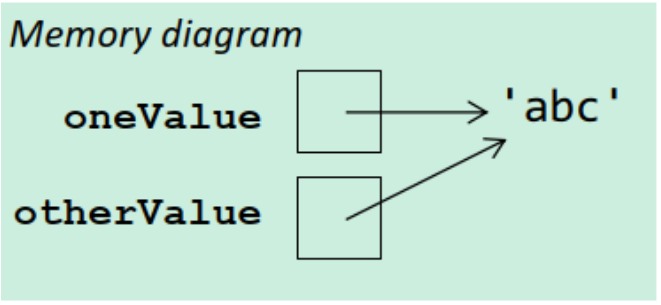
The inputs to a function are called its arguments and the function is said to be called on its arguments. In Python, the arguments in a function call are delimited by parentheses and separated by commas.

Understanding variable and function names

One value can have multiple names. These names refer to the same value in the computer memory. See the examples below for variables and functions.

```
>>> oneValue = 'abc'

>>> otherValue = oneValue

>>> oneValue
'abc'

>>> otherValue
'abc'
```

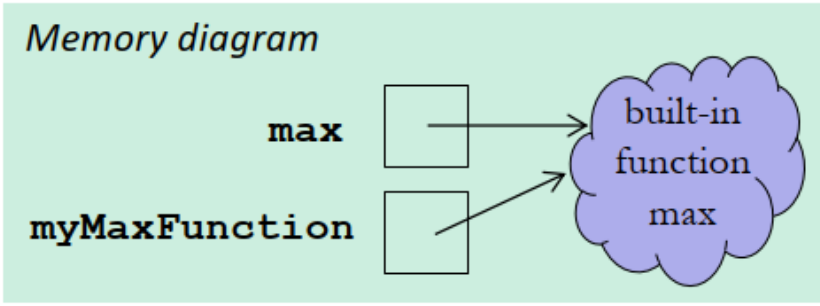
Memory diagram



Functions are values. Just like numbers & strings

```
>>> max
<built-in function max>

>>> myMaxFunction = max

>>> max(10,100)
100

>>> myMaxFunction(10,100)
100
```

Memory diagram



Built-in functions: id

This function displays the memory address where a value is stored.

```
>>> id(oneValue)
4526040688

>>> id(otherValue)
4526040688
```

Different names can refer to the same value in memory.

```
>>> id(max)
4525077120

>>> id(myMaxFunction)
4525077120
```

Built-in functions: type

- **Each Python value has a type**. It can be queried with the built-in type function.
- Types are special kinds of values that display as <class 'typeName'>
- Knowing the type of a value is important for reasoning about expressions containing the value.

In [...]	Out [...]
`type(123)`	`int`
`type(3.141)`	`float`
`type(4 + 5.0)`	`float`
`type('CS111')`	`str`
`type('111')`	`str`
`type(11/4)`	`float`
`type(11//4)`	`int`
`type(11%4)`	`int`
`type(11.0%4)`	`float`
`type(max(7, 3.4))`	`int`
`x = min(7, 3.4)`	`# x gets 3.4`
`type(x)`	`float`
`type('Hi,' + 'you!')`	`str`
`type(max)`	`builtin_function_or_method`
`type(type(111))`	`type # Special type for types!`

Built-in functions: type

```
>>> type(10)
<class 'int'>
>>> type('abc')
<class 'str'>
>>> type(10/3)
<class 'float'>
>>> type(max)
<class 'builtin_function_or_method'>
>>> type(len)
<class 'builtin_function_or_method'>
>>> type(True)
<class 'bool'>
>>> type([1,2,3])
<class 'list'>
>>> type((10,5))
<class 'tuple'>
```

Functions are values with this type

Other types we will learn about later in the semester

Built-in functions: len

When applied to a **string**, the built-in `len` function returns the number of characters in the string.
`Len` raises a `TypeError` if used on values (like numbers) that are not sequences. (We'll learn about sequences later in the course.)

In [...]	Out [...]
`len('CS111')`	`5`
`len('CS111 rocks!')`	`12`
`len('com' + 'puter')`	`8`
`course = 'computer programming'`	
`len(course)`	`20`
`len(111)`	`TypeError`
`len('111')`	`3`
`len(3.141)`	`TypeError`
`len('3.141')`	`5`

Built-in functions: str

The **str** built-in function returns a string representation of its argument.
It is used to create string values from **int**s and **float**s (and other types of values we will meet later) to use in expressions with other string values.

In [...]	Out [...]
`str('CS111')`	`'CS111'`
`str(17)`	`'17'`
`str(4.0)`	`'4.0'`
`'CS' + 111`	`TypeError`
`'CS' + str(111)`	`'CS111'`
`len(str(111))`	`3`
`len(str(min(111, 42)))`	`2`

Built-in functions: int

- When given a string that's a sequence of digits, optionally preceded by +/-, **int** returns the corresponding integer. On any other string it raises a **ValueError** (correct type, but wrong value of that type).
- When given a float, **int** return the integer the results by truncating it toward zero.
- When given an integer, **int** returns that integer.

```
In [...]                Out [...]
int('42')               42
int('-273')             -273
123 + '42'              TypeError
123 + int('42')         165
int('3.141')            ValueError  # strings are not sequence
int('five')             ValueError  # of chars denoting integer
int(3.141)              3
int(98.6)               98          # Truncate floats toward 0
int(-2.978)             -2
int(42)                 42
int(-273)               -273
```

Built-in functions: float

- When given a string that's a sequence of digits, optionally preceded by +/-, and optionally including one decimal point, `float` returns the corresponding floating point number. On any other string it raises a `ValueError`.
- When given an integer, `float` converts it to floating point number.
- When given a floating-point number, `float` returns that number.

In [...]	Out [...]
`float('3.141')`	`3.141`
`float('-273.15')`	`-273.15`
`float('3')`	`3.0`
`float('3.1.4')`	`ValueError`
`float('pi')`	`ValueError`
`float(42)`	`42.0`
`float(98.6)`	`98.6`

Built-in functions: round

- When given **one** numeric argument, `round` returns the **integer** it's closest to.
- When given **two** arguments (a numeric argument and an integer number of decimal places), `round` returns **floating point** result of rounding the first argument to the number of places specified by the second.
- In other cases, `round` raises a `TypeError`

```
In [...]                    Out [...]
round(3.14156)              3
round(98.6)                 99
round(-98.6)                -99
round(3.5)                  4
round(4.5)                  5
                            # always rounds up for 0.5
round(2.718, 2)             2.72
round(2.718, 1)             2.7
round(2.718, 0)             3.0
round(1.3 - 1.0, 1)         0.3
round(2.3 - 2.0, 1)         0.3
                            # Compare to previous slide
```

Built-in functions: `print`

`print` displays a character-based representation of its argument(s) on the screen and **returns** a special **`None`** value.

Note that **`print`** also does **not** display any quotation marks for strings.

Console

```
print(7)
print('ME444')
print(len(str('ME444')) * min(17,3))
college = 'Gaziantep'
print('I go to ' + college)
dollars = 10
print('The movie costs $'
+ str(dollars) + '.')
```

```
7
ME444
15
I go to Gaziantep University
The movie costs $10.
```

The newline character `'\n'`

`'\n'` is a single special **newline character. Printing it causes the console to shift to the next line.**

In [...]

```
print('one\ntwo\nthree')
```

Console

```
one
two
three
```

`Print` with multiple arguments

When `print` is given more than one argument, it prints all arguments, separated by one space by default. This is helpful for avoiding concatenating the parts of the printed string using `+` and using `str` to convert non strings to strings.

In [...]

```
print(6,'*',7,'=',6*7)

# print with one argument is much
# more complicated in this example!
print(str(6)+' * '+str(7)+' = '+str(6*7))
```

Console

```
6 * 7 = 42
6 * 7 = 42
```

`Print` with the `sep` keyword argument

`Print` can take an optional so-called *keyword argument* of the form `sep=`*stringValue* that uses *stringValue* to replace the default space string between multiple values.

In [...]

```
print(6,'*',7,'=',6*7)

# replace space by $
print(6,'*',7,'=',6*7,sep='$')

# replace space by two spaces
print(6,'*',7,'=',6*7,sep='  ')

# replace space by zero spaces
print(6, '*',7,'=',6*7,sep='')

# replace space by newline
print(6,'*',7,'=',6*7,sep='\n')
```

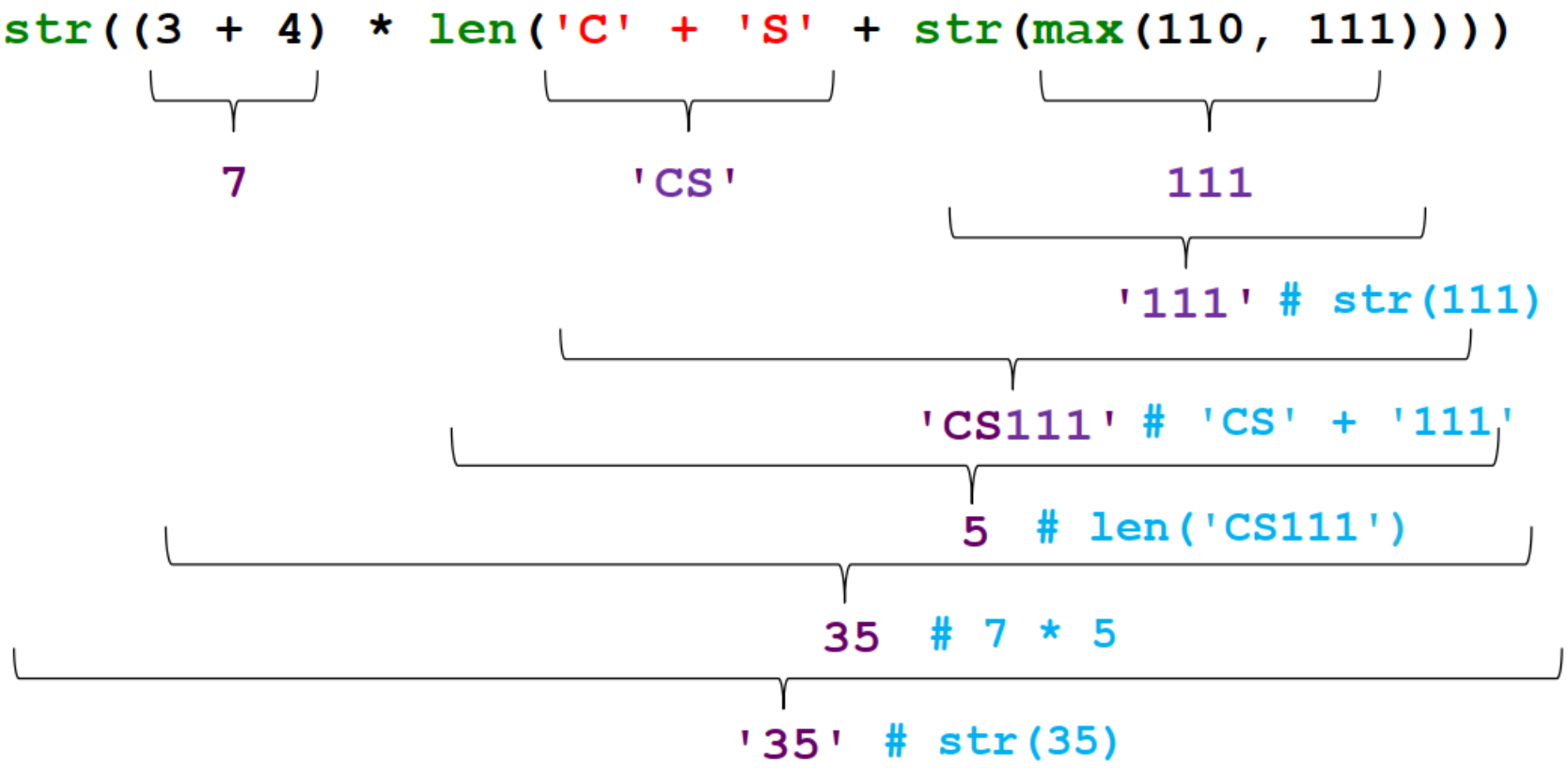
Console

```
6 * 7 = 42

6$*$7$=$42

6  *  7  =  42

6*7=42

6
*
7
=
42
```

Complex Expression Evaluation

o An **expression** is a programming language phrase that denotes a value. Smaller **sub-expressions** can be combined to form arbitrarily large expressions.

o Complex expressions are evaluated from "inside out", first finding the value of smaller expressions, and then combining those to yield the values of larger expressions. See how the expression below evaluates to `'35'`:



More `print` examples

```
In [4]: print('one\ntwo\three')  # '\n' is a single special
                                 # newline character.
one                              # Printing it causes the
two                              # display to shift to the
three                            # next line.

In [5]: print('one', 'two', 'three', sep='\n')
                                 # Like previous example,
one                              # but use sep keyword arg
two                              # for newlines
three

In [6]: str(print(print('CS'), print('CS')))
CS # printed by 2nd print
111 # printed by 3rd print.
None None # printed by 1st print; shows that print returns None

Out [6]: 'None' # result of str; shows that print returns None
```

More `print` examples

```
message="Welcome to ME444"

message
'Welcome to ME444'

print(message)
Welcome to ME444

print(10+20)
30

print(max(10,20))
20
```

```
max(10,20)
20
a=print(max(10,20))
20
a+20
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    a+20
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

Be careful, **print** converts the arguments to **string**!

Built-in functions: `input`

`input` displays its single argument as a prompt on the screen and waits for the user to input text, followed by Enter/Return. It returns the entered value as a **string**.

```
In [7]: input('Enter your name: ')
Enter your name: Olivia Rodrigo

In [8]: age = input('Enter your age: ')
Enter your age:20

In [9]: age
Out [9]: '20'

In [10]: age + 4
TypeError
```

No output from assignment.

Value returned by **`input`** is always a **string**. Convert it to a numerical type when needed.

Tried to add a string and a float.

Built-in functions: `input`

```
In [11]: age = float(input('Enter your age: '))
Enter your age: 18

In [12]: age + 4
Out [12]: 22.0
```

Example of nested function calls.

age contains **float('18')**, which is **18.0** and **18.0 + 4** is **22.0**

Expressions vs. Statements

Phrases that **produce a value**. E.g. :

```
10
10 * 20 - 100/25
max(10, 20)
int("100") + 200
fav
fav + 3
"pie" + " in the sky"
```

Expressions are composed out of any combination of values, variables operations, and function calls.

Phrases that **perform an action** / **change the state of the program** (can be visible, invisible, or both):

```
print(10)
age = 19
teleport(0, 150)
```

Statements may contain expressions, which are evaluated **before** the action is performed.

```
print('She is ' + str(age)
+ ' years old.')
```

We'll consider expressions that return a `None` value to be kinds of statements.

Putting Python code in a .py file

Rather than interactively entering code into the **Python Shell**, we can enter it in the **Editor Pane**, where we can edit it and save it away as a file with the **`.py`** extension (a Python program). Here is a **`nameage.py`** program. Lines beginning with **`#`** are comments We run the program by pressing the triangular "run"/play button.

a.py - C:/Users/N.Furkan/Desktop/a.py (3.13.7)

File Edit Format Run Options Window Help

```
#Part 1: Gather personal informations
name=input('Enter your name:')
age=input('Enter your age:')
#Convert age from str to integer
age=int(age)

#Part 2: Output the results by concantaneting the strings with variab
#values that store user data.

print("Hello, "+ name + ".")
print("In 4 years, you'll be "+str(age+4)+" years old.")
```

Ln: 11 Col: 56

→

```
Enter your name:Furkan
Enter your age:37
Hello, Furkan.
In 4 years, you'll be 41 years old.
```

Error messages in Python

Type Errors

`'111' + 5` **TypeError**: cannot concatenate 'str' and 'int' values

`len(111)` **TypeError**: object of type 'int' has no len()

Value Errors

`int('3.142')` **ValueError**: invalid literal for int() with base 10: '3.142'

`float('pi')` **ValueError**: could not convert string to float: pi

Name Errors

`CS + '111'` **NameError**: name 'CS' is not defined

Syntax Errors

A syntax error indicates a phrase is not well formed according to the rules of the Python language. E.g. a number can't be added to a statement, and variable names can't begin with digits.

```
1 + (ans=42)
1 + (ans=42)
      ^
SyntaxError: invalid syntax
```

```
2ndValue = 25
2ndValue = 25
^
SyntaxError: invalid syntax
```

Test your knowledge

1. Create simple **expressions** that combine **values** of different **types** and math **operators**.
2. Which operators can be used with **string values**? Give examples of expressions involving them. What happens when you use other operators?
3. Write a few **assignment statements**, using as assigned values either **literals** or expressions. Experiment with different **variable names** that start with different characters to learn what is allowed and what not.
4. Perform different **function calls** of the **built-in functions**: `max`, `min`, `id`, `type`, `len`, `str`, `int`, `float`, `round`.
5. Create **complex expressions** that combine variables, function calls, operators, and literal values.
6. Use the function `print` to display the result of expressions involving string and numerical values.
7. Write simple examples that use `input` to collect values from a user and use them in simple expressions. Remember to **convert** numerical values.
8. Create situations that raise different kinds of **errors**: `TypeError`, `ValueError`, `NameError`, and `SyntaxError`.